# EMC® Documentum®
# Dynamic Delivery Services
**Version 6.5 SP1**

## User Manual
## P/N 300-008-589-A01

EMC Corporation
*Corporate Headquarters:*
Hopkinton, MA 01748-9103
1-508-435-1000
www.EMC.com

Copyright ©2008 EMC Corporation. All rights reserved.

Published December 2008

EMC believes the information in this publication is accurate as of its publication date. The information is subject to change without notice.

THE INFORMATION IN THIS PUBLICATION IS PROVIDED AS IS. EMC CORPORATION MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND WITH RESPECT TO THE INFORMATION IN THIS PUBLICATION, AND SPECIFICALLY DISCLAIMS IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Use, copying, and distribution of any EMC software described in this publication requires an applicable software license.

For the most up-to-date listing of EMC product names, see EMC Corporation Trademarks on EMC.com.

All other trademarks used herein are the property of their respective owners.

# Table of Contents


**Preface** ..... 3
- Purpose of this manual ..... 3
- Conventions ..... 3
- Revision history ..... 4
- Getting information ..... 4

**Introduction** ..... 5
- Features ..... 5
- Benefits for existing customers ..... 7
- Getting familiar with the product ..... 7
- Architecture ..... 7

**Administration** ..... 10
- Included web-applications ..... 10
- The Admin application ..... 11
- Uploading data into the XML Store ..... 11
- XML Store Data Layout ..... 11
- XML Store Builtin Metadata ..... 13
- A Development Process Example ..... 14
- A test/deployment example ..... 16
- DDS Ant targets ..... 17
- DDS Ant tasks ..... 21

**API, Frameworks and Services** ..... 25
- Core concepts ..... 25
- Server-side Application Development ..... 27
- Adding Server Functionality ..... 29
- Persistence Layer ..... 30
  - Using the Persistence Layer ..... 36
  - Persistence Layer and XDB ..... 38
  - Persistence Layer and File System ..... 38
  - Persistence Layer and XAM ..... 39
- Application Framework ..... 40
  - Application Configuration ..... 42
- Structures Framework ..... 48
  - Store Locations ..... 50
  - DDS Locales and Java locales ..... 51
  - Storage examples ..... 51
- Services Framework ..... 52
  - Services Lifecycle ..... 55
- Operation Framework ..... 57
- User Service ..... 61
- Response Service ..... 63
- Logging Framework ..... 66
- XBase Framework ..... 68
- XProc Service ..... 70
  - XProc Service Configuration ..... 72

Logic Engine Service ... 73
Logic Engine Service Configuration ... 76
DDS URIs ... 77

**Client Services** ... 80
GWT Services ... 80
GWT Widget Library ... 84
GWT Application Service ... 85
GWT User Service ... 86
GWT Log Center Service ... 86
GWT Persistence Service ... 87
GWT XML Persistence Service ... 87
GWT Resource Service ... 88
GWT Index Service ... 89
GWT XQuery Service ... 89
GWT I18N Service ... 90
GWT XProc Service ... 90
GWT Logic Engine Service ... 91
DDS Tag Library ... 94

**DDS Admin Client** ... 99
The Admin user interface ... 99
Navigation ... 100
Using dialogs ... 100
Applications and Data ... 101
Working with Applications ... 102
Creating an application ... 103
Renaming an object ... 103
Deleting an object ... 103
Working with libraries ... 104
Creating a library ... 104
Adding a BLOB document ... 105
Uploading a ZIP file ... 105
Working with XML documents ... 105
Adding an XML document ... 106
Editing an XML document ... 106
Viewing an XML document ... 106
Working with Existing Indexes ... 107
Deleting indexes ... 108
Working with Index suggestions ... 108
Data Analysis ... 109
Creating an Index ... 110
Changing Index suggestion options ... 111
Resetting Index suggestions ... 112

**DEMO APPLICATIONS** ... 113
Garage demo ... 113
Using the application ... 114
DDS Kitchensink demo ... 118
Logic Engine demo ... 120
Taglib demo ... 123

# Preface

## Purpose of this manual

This manual provides information about EMC Documentum Dynamic Delivery Services (DDS).

The manual contains the following chapters:

- **Introduction**: gives a short introduction to the product
- **Administration**: information on administration and development of applications
- **API, Frameworks and Services**: describes main concepts and server features of the product
- **Client Services**: describes client features of the product
- **Admin client**: an introduction to using the Admin web client
- **Demo Applications**: an introduction to using some of the included demo applications

**Intended audience**

This manual is written for application developers and system administrators who want to build XML delivery applications. It assumes familiarity with the concepts of document processing. It also assumes a working knowledge of web-based software development, JAVA and XML.

## Conventions

This document uses the following conventions.

**Conventions**

| Convention | Meaning |
|---|---|
| Initial capital letter | In DDS terms and concepts, words start with a capital letter. For example:<br>DDS Application<br>Persistence Layer |
| `example code` | `Program code and commands, used` in examples:<br>`String pipelineURI = ...;`<br>`String dataURI = ...;`<br>`boolean readOnly = true;`<br>`// Get the GWT XProcService instance` |

| Convention | Meaning |
|---|---|
| `filenames and pathnames` | File names and file paths, for example:<br>Go to the `../DDS/bin` folder.<br>Set values in the `.properties` files. |

# Revision history

The following changes have been made to this document:

**Revision history**

| Revision | Description |
|---|---|
| April 2008 | First publication. |
| December 2008 | Service Pack 1 |

# Getting information

**Document files**
Files containing information about included software, licensing, and API usage can be accessed in folder `DDS/doc`. These files include:

| File | Description |
|---|---|
| eula.txt<br>third-party licenses.txt | License agreements |
| api/index.html | API documentation for developers. |

# Introduction

## Features

**Dynamic Delivery Services (DDS)**

EMC Documentum Dynamic Delivery Services (DDS) provides means for producing XML delivery applications with minimal development effort, running in a web browser, served by a scalable delivery platform based on EMC Documentum xDB.

DDS is suited to organizations that need to deliver large and varied quantities of personalized content.

Below is a summary of main product features.

**Features of Dynamic Delivery Services**

DDS Service Pack 1 introduces several new features:

- File System support.

  Windows and UNIX file systems can be used for data storage.

- Support for EMC Centera.

  XAM (eXtensible Access Method) can be used for data storage, for example with EMC Centera systems.

- Support for metadata storage and retrieval.
- Support for dealing with information supplied by end-users

  A new User Supplied Information (USI) framework can be used for dealing with the reception and processing of information supplied by end-users, for example comments, votes, ratings etc.

Features already present in previous DDS versions include:

- Easy to install and use.

  After installation, take a few hours for familiarization, and your first basic XML delivery application can be up and running from day one.

  Generated end-user applications are 100% Java, run anywhere.

- Based on open standards and open source: XML, XQuery, XProc, XForms, Java, Apache Tomcat, Google Web Toolkit.
- Provides a rich API to developers.

  The API offers support for server-side and client-side development.

- Leverages high-performance XML database.

  xDB offers reliable and scalable XML data storage, with fast access regardless of the number of concurrent users, or the number of documents or the database size. It has a built-in transaction mechanism, and supports load balancing and replication of databases over multiple machines.

  DDS' extensive server-side framework makes it easy to use existing and compose new services based on xDB capabilities.

  DDS' admin application also assists you in having the xDB maintain the appropriate indexes on your data for optimal retrieval performance.

- Includes GWT widgets and services.

  DDS comes with an extensive set of GWT widgets and services tailored to be used in combination with the DDS services.

- Includes a library of JSP tags.

  A tag library facilitates the use of DDS services from JSP-based applications.

- Included source code samples.
- Support for synchronization with EMC Documentum Content Server.

  DDS can be used alongside Documentum Content Server, using data feeds from its Site Caching Services (SCS). SCS is part of the EMC Documentum Site Delivery Services product set, and prefetches content and metadata to a standards-based repository.

- Support for EMC Documentum Forms Builder.

**XProc Processor**

DDS comes with a built-in XProc processor. XProc, or An XML Pipeline Language, is a language for describing operations to be performed on XML documents. An XML Pipeline specifies a sequence of operations to be performed on one or more XML documents. Pipelines generally accept one or more XML documents as input and produce one or more XML documents as output. Pipelines are made up of simple steps which perform atomic operations on XML documents and constructs similar to conditionals, loops and exception handlers which control which steps are executed.

**XForms Engine**

DDS offers a completely client-side XForms Engine. The engine is capable of processing and rendering forms, in GWT applications as well as clients based on different technologies (e.g. JSP).

**Logic Engine**

DDS includes an S1000D Process Data Module Logic Engine implementation. The S1000D Process Data Module is a standard XML format for representing interactive structures such as maintenance or troubleshooting procedures, or various "wizard-like" interfaces.

The Logic Engine is a software component that represents an interpreter of Process Data Modules. It provides an interactive interface to the end-users and performs actions such as applicability filtering, branching and looping, to guide end-users through the process defined in the Process Data Module, ensuring that requisite steps in a procedure or process are followed in proper order. The interface can help the end-user by supplying additional information and by interfacing to external systems or devices, as well as by producing PDF documents.

# Benefits for existing customers

**Documentum Content Server customers**

Owners of an EMC Documentum Content Server Platform can conveniently use Site Caching Services (SCS) to incrementally publish content and metadata to the storage facilities included in EMC Documentum Dynamic Delivery Services (DDS), for use by DDS-generated delivery applications.

SCS has been enhanced to offer the ability to install and configure xDB on target host machines.

**X-Hive/DB customers**

Organizations having solutions based on XML Store, or on X-Hive/DB 8, can simply integrate DDS into their current system, significantly enhancing their publishing capabilties.

# Getting familiar with the product

As a user, you should know how to use the product for your own purposes. This will be easier if you understand the concepts that apply to your situation.

Before you start using the product, you should know how to develop web applications using Java and XML.

**Background information**

The product is developed in Java, and uses EMC Documentum xDB as XML Store.

Third-party software required for development includes:

- Apache Tomcat
- Apache Ant
- Google Web Toolkit (GWT)
- Java Development Kit (JDK)

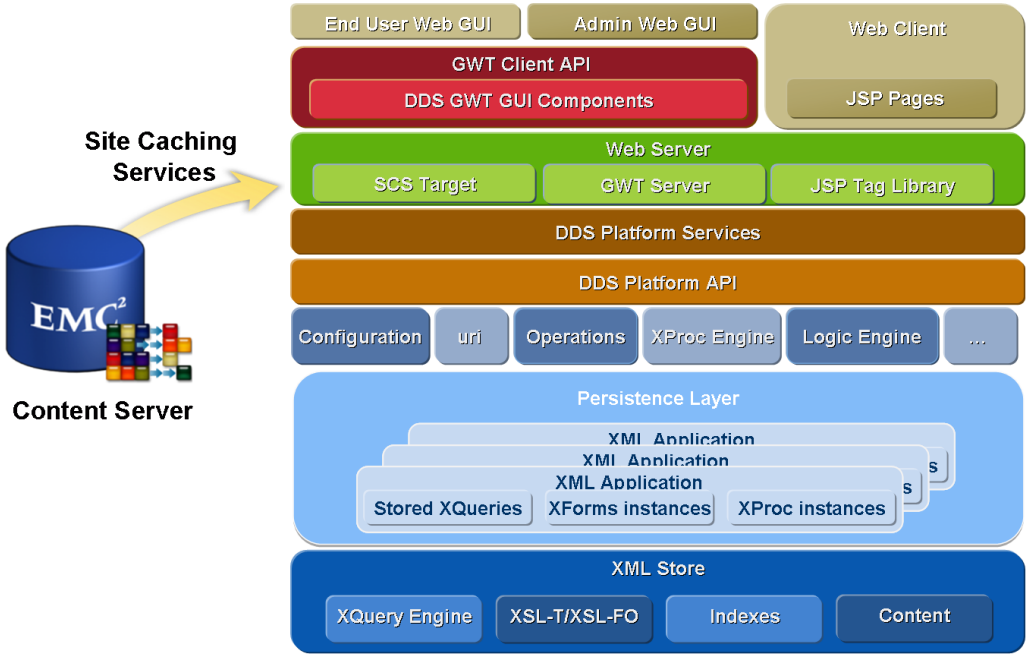
See the Release Notes for more information about hardware and software requirements.

As an aid to developers, the product includes several sample applications.

# Architecture

DDS has a layered architecture, with each layer using the functionality (through the exposed API) of the ones below.

The layers are organized as shown in the following figure:




### XML Store

For a DDS application, the XML Store may hold all application-specific resources (XProc instances, XForms, Process Data Modules, XSLT Stylesheet, etc.), as well as all of the data on which the application works (a repository of XML content, images, audio files, etc.). APIs offer access to functionality such as data storage and retrieval, transaction management, indexing, querying, etc.

### Persistence Layer

The DDS Persistence Layer provides abstractions which allow the DDS developer to persist, retrieve and manage data without reference to the type of the underlying storage facilities (e.g. an XML Store, a file system, or a Documentum repository).

### Frameworks

On top of the persistency layer are a number of frameworks, the most important of which are:

- Application Configuration , for configuring an application in terms of stores used, services used, users, authentication, etc.
- Operation Framework , for executing server side functionality in a controlled manner (i.e. with, among others automatic session management, and transaction support)
- URIs , for providing access to all persistent content through URIs
- XProc Engine, for interpreting and executing XProc pipelines
- Logic Engine, for interpreting and executing Process Data Modules

### DDS API and Services

The Persistence Layer and the Frameworks together provide the DDS Platform's API. Some of its functionality is exposed to the developer in a more controlled way through the DDS Platform Services (e.g. UserService, XProcService, TokenService).

### Web Server

On top of the core is the layer adapting DDS core services to specific clients or client types.

- Through the SCS Target, data from a Documentum Content Server can be imported into a DDS store.
  **Note:** The SCS Target is actually not shipped with DDS, but with Documentum SCS.
- An extensive set of GWT services is available to GWT-based DDS clients

- For JSP-based clients, DDS offers a tag library of convenient DDS specific functionality.

**GWT Client**

On the client side, DDS offers a GWT widgets set, which can be used with the GWT core widgets in a browser application. Interaction with the server-side of DDS is through the GWT services of the previous layer.

**Application Development**

In principle, all APIs of all DDS layers are available to the application developer. From the sections on application development, demo applications, and the API, developers can learn on what layers best to develop. For example, if a new GWT-based application requires some intricate piece of persistence logic on the server side, which is not offered as such by DDS, the developer may find that a new Operation must be devised, which can then be made available as a GWT service. Alternatively, he/she may find that combining existing Operations suffices, and that the only thing needed is to write the GWT service using them.

# Administration

## Included web-applications

DDS comes with several web applications, including an Administrator tool and several demo applications using a different user interface, different data or both.

For instructions on building and running the demo applications, see the Installation Guide.

By running demo applications, and by examining the included source code, a web application developer can get a good, basic impression of the kind of results that can be achieved with DDS.

**Admin**

A basic DDS front-end, this Administrator tool is intended for general administration and development of DDS applications.

**Garage**

A demo application with a graphical user interface, including:

- menus and tabs
- a Table of Contents tree
- switching between datasets
- content searching
- content assembly and delivery
- output to PDF and HTML

DDS applications can provide grab and bind functionality to end-users, allowing them to compose their own virtual documents from selected content documents or fragments. These virtual documents can be printed, or output as PDF or HTML to computer screens and handheld devices by means of drag-and-drop.

The garage demo uses several different data sets, including one based on the DITA garage toolkit and another based on The Project Gutenberg EBook of Encyclopaedia Britannica, 11th Edition, Volume 4, Part 3. Garage shows a possible way to present content in a web application, including use of menus, window panels, folders, search options and publications.

**Kitchen sink**

An extension of the GWT Kitchen Sink sample, this demo application shows general behavior of services and widgets.

**Taglib**

A demo application to show usage of tag libraries for data retrieval from an XML Store. It uses the data from the garage demo application.

**Logic engine**

DDS comes with a built-in S1000D Process Data Module Logic Engine, a software component that executes S1000D Process Data Modules and provides interactivity with the user. To show usage of the Logic Engine, a demo application is included.

The S1000D Process Data Module specification is a standard for interactive processing structures. See http://s1000d.org for more information about S1000D.

# The Admin application

DDS comes with a web application (see admin ) for use with basic application admininstration and development work, including:

- Manage applications:
  - Create, rename and delete applications
  - Add, rename, delete and browse application libraries
  - Add, edit, rename and delete XML documents in libraries
  - Add, rename and delete non-XML documents in libraries
- Browse data sets and manage indexes

See the DDS Installation Guide for information about building, deploying and running **Admin**.

# Uploading data into the XML Store

Content for DDS applications can be uploaded into the XML Store by means of:

- DDS Ant tasks
- the DDSDataImporter Java API

From EMC Documentum, data can be provided to DDS through the EMC Documentum Site Caching Server.

# XML Store Data Layout

There are two distinct groups of data in DDS:

1. Data Sets, which are composed of actual (publishable) content data and its metadata.
2. Application Data, composed of scripts such as XQueries, stylesheets, XProcs, XForms etc.

**Application Data**

Each DDS application should have its own library inside the library "/APPLICATIONS".

For example, with two applications, called "admin", and "garage", you would have:

- `/APPLICATIONS/admin`
- `/APPLICATIONS/garage`

Inside an application library, there are several predefined libraries, for specific purposes:

- `/APPLICATIONS/<applicationName>/configuration` holds configuration files such as the Services, XBase and Store configuration files.
- `/APPLICATIONS/<applicationName>/resources` is for storing application resources such as XQueries, stylesheets, XProcs, XForms etc.

- `/APPLICATIONS/<applicationName>/users` hosts user-specific configuration for DDS Users as well as their "home libraries".

The configuration library typically contains the Services.xml, Structures.xml, Stores.xml, XBases.xml and other configuration files.

The resources library typically has several child libraries, one for each type of resource:

- `/APPLICATIONS/<applicationName>/resources/xforms`
- `/APPLICATIONS/<applicationName>/resources/xproc`
- `/APPLICATIONS/<applicationName>/resources/xslt`

The users library has one XML file containing the configuration for the DDS User, and one library, which acts as the home library for that User. Any documents specific to that User should be stored in the home library. The name of the configuration file is the User Id with the ".xml" suffix added. The name of the home library is simply the User Id.

Each XForms script should have its own library inside `/APPLICATIONS/<applicationName>/resources/xforms`. This is necessary because XForms scripts can consist of multiple files.

**Data sets**

Each data set is stored in a single library named after the dataset, which should be placed inside the library "DATA". So a data set called "garage" would be stored in a library /DATA/garage.

Additionally, a configuration file with settings for the data set is created in the same location. For a data set called "garage", this would be called garage.xml, in the /DATA library.

A data set is assumed to be composed of two groups of files: the actual content files, and their metadata files. The content files are stored inside a sub-library called "Collection", and the metadata files are stored inside a sub-library called "CollectionMetadata". So if there are two data sets, say "garage", and "britannica", their libraries would be:

- `/DATA/garage/Collection`
- `/DATA/garage/CollectionMetadata`
- `/DATA/britannica/Collection`
- `/DATA/britannica/CollectionMetadata`

If the data set contains files from multiple locales, each locale in the collection will add another sub-library to the previous structure. The name of this sub-library is the name of the locale itself. So if the "britannica" dataset had two locales, "en_IN" and "fr_FR", that would give

- `/DATA/britannica/Collection/en_IN`
- `/DATA/britannica/Collection/fr_FR`
- `/DATA/britannica/CollectionMetadata/en_IN`
- `/DATA/britannica/CollectionMetadata/fr_FR`

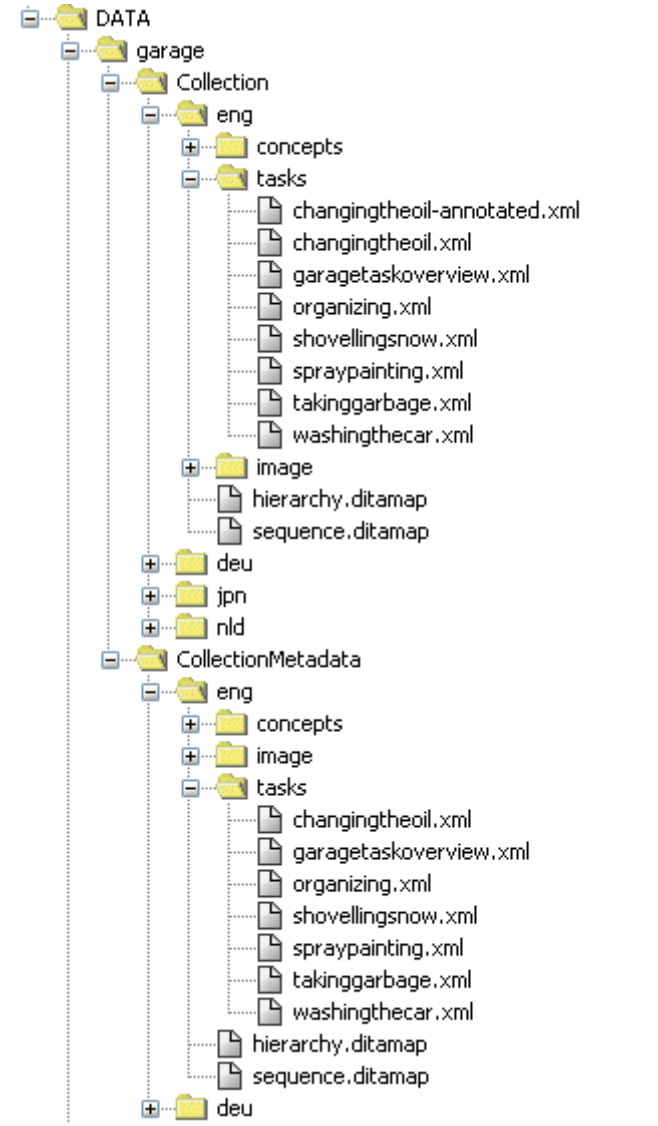
If no locale is set, then the files are to be stored in the root of the "Collection", and of the "CollectionMetadata".

**Metadata**

DDS expects the metadata of content files to be in XML format. The XML Store has a built-in metadata, but as this only supports key/values pairs, it cannot be used to store the DDS metadata, which is assumed to be in XML format. Therefore the metadata of content files is always stored in a separate XML Document.

As we have seen, the content data and its metadata are stored in two separate libraries Collection and CollectionMetadata. Under these, we will see identical parallel structures of libraries and files

containing data, and libraries and files containing metadata, as shown in the following screenshot of the xDB administrator, after loading the data of the Garage demo application:




# XML Store Builtin Metadata

The XML Store holds builtin metadata on data sets.

**dds:subscription-element**

Each file in DDS is expected to have a unique identifier, called `subscription-element`. This is stored in XML Store Builtin Metadata, both for items in Collection, and in CollectionMetadata, and serves as the link between a metadata file inside CollectionMetadata and its content file inside Collection.

The content data and its metadata are stored in parallel structures, so why do we need an XML Store Builtin Metadata field to link the content entry with its metadata? Because, if a folder has metadata, and content files inside of it, we cannot have an XML Document in the same path in the

"CollectionMetadata". In such a case the metadata of the folder will be placed inside a nameless file inside the folder, and linked back to the content file through the "dds:subscription-element" key.

**dds:locale**

The locale to which each file belongs is also set in the XML Store Builtin Metadata through the key dds:locale.

**dds:content-path**

The original content path of each file inside a data set is stored in the XML Store Builtin Metadata through the key "dds:content-path".

# A Development Process Example

Developers of web applications can use DDS in various different ways.

A typical web application will include:

- content in an XML database
- a web application tailored to run on top of that database

DDS provides for an XML store to hold the data, and the means to construct a basic application frame and user interface.

A general approach to development is outlined below. This example assumes use of a command-line tool to execute Ant tasks in the `../bin` folder of a standard DDS installation on a local computer (including JDK, GWT and Tomcat) by a single developer.

1. Ensure that **Tomcat** is not running.
2. Go to the `DDS/bin` folder.
3. Set values in the `.properties` files, including ant-environment.properties and ant-xdb.properties.

   **Note:** Values used in the steps below should match those in the properties files.

4. If the application requires a new database:
   a. Create a new, empty database.

      ```
      dds-ant create-database
        -Dname=${your.database.name}
        -Dadmin.password=${your.admin.password}
        -Dsuperuser.password=${your.superuser.password}
      ```

   b. Create the database layout.

      ```
      dds-ant create-database-layout
      ```

      This creates the required APPLICATIONS and DATA libraries in the database (see Data layout ).

   Alternatively, you can use the admin tool of the XML Store to create the database and its layout on the federation.

5. Start **Tomcat (using a separate command-line session)**.
6. Create database structures.

a. Create the application.

```
dds-ant create-application -Dapplication=${your.application.name}
```

This will create a library structure under /APPLICATIONS in the database, with the name of your application.

b. Create the dataset(s).

```
  dds-ant create-dataset -Ddataset=${your.dataset.name}
```

This will create a library structure in the database under /DATA with ${your.dataset.name} as its root.

c. Create the library structure.

```
  dds-ant create-template -Dapplication=${your.application.name}
```

Alternatively, you can use the DDS admin application.

This Ant task creates a folder/file structure under ../applications/${your.application.name} that is suitable for use with the DDS Ant tasks:

- bin
  (contains build.xml.template, build.properties.template)
  **Note:** If you want to use a template file, just remove its ".template" extension and edit it as required.
- data

  place any data here that is to be loaded into the database (possibly in a subfolder application, for application-specific database content such as XProc and XForms documents)
- lib

  place any jar files here that are needed on the server side of a deployed application
- resources

  place any resources here that you need to be on the classpath of a deployed web application (these may include XProc an XForms documents, but also any other stuff, such as configuration files, language-specific resources etc.)
- src

  develop the application's GUI here, in terms of a composition of interacting widgets and services
- war/WEB-INF (with web.xml.template)
  **Note:** The web.xml.template contains all DDS-specific servlet mappings. You can add your own to this file.

7. Load application data.
   a. Load application-specific data, for example XForms and Xprocs, into the database (usually under /APPLICATIONS/${your.application.name).

```
dds-ant load-application-data
-Dapplication=${your.application.name}
```

b. Load the dataset(s) (usually under /DATA/{some.dataset.name}.

```
dds-ant load-data-sets -Dapplication=${your.application.name}
```

Alternatively, you can load both application data and its data sets into the database with a single Ant task:

```
dds-ant load-all-data -Dapplication=${your.application.name}
```

8. Develop your application, including client and server side java code, using DDS and other development tools as required.

   For GWT, follow its recommended client/server package structure, as applied in the sample applications kitchensink and garage.

9. Test the application, by creating a war and deploying it (see below), and/or by running it in GWT hosted mode.
   - to run in GWT hosted mode:.

```
dds-ant run -Dapplication=${your.application.name}
```

   **Note:** Use deployed mode for performance testing. In hosted mode, an application will typically run considerably slower.

10. If the application runs without errors, you can create a war and deploy it for use.
    a. Create a war.

```
dds-ant create-war -Dapplication=${your.application.name}
```

       This will create the .war file of the application in `../build/`.

    b. Deploy the .war file to the target Tomcat server, for example using the Tomcat manager tool.
    c. Test-run the deployed application.

       If the target Tomcat server is installed locally, and properly configured, the web application is now available at `http://localhost:8080/{name}`.

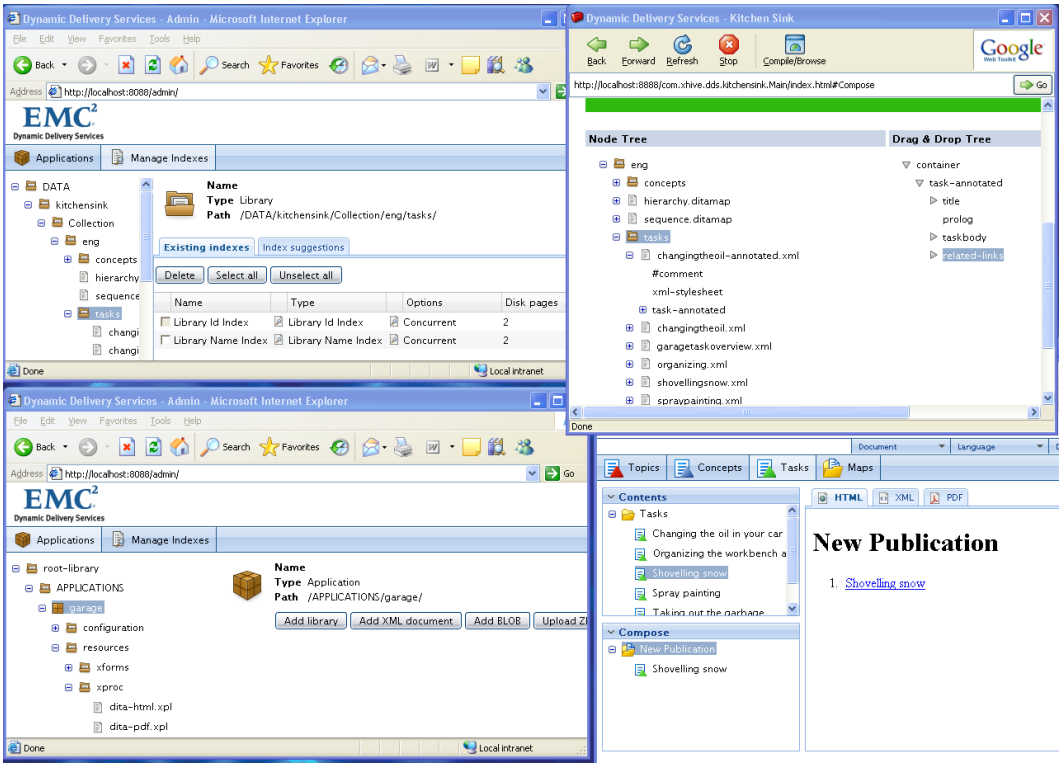
# A test/deployment example

A typical approach is for developers to test-run their applications in GWT hosted mode during the early stages of development, and to test more and more in deployed mode as application development advances.

A single developer can be working in parallel on several related applications, which may be in different stages of development.

The example below shows a DDS Admin client at top-left, displaying the Kitchen Sink data set, with the Kitchen Sink application running in GWT hosted mode right beside it.

Another DDS Admin at bottom-left simultaneously shows the garage application library, with garage running in deployed mode besides it at bottom-right.




In the garage application, the user has just started a new publication and dragged the first item into it.

# DDS Ant targets

Administrators and Developers can use DDS Ant targets for various command line driven tasks, including:

- Uploading volumes of data into the XML Store (also data coming from SCS)
- Setting up application and dataset structures in the XML Store
- Setting up a template folder structure before developing a new application
- Compiling, deploying and running developed applications

Ant targets in build.xml include:

- clean
- create-template
- create-database
- create-database-layout
- delete-dataset
- create-application
- delete-application
- build
- load-application-data
- load-data-sets
- load-all-data
- run
- create-war

- run-xforms-builder

A typical DDS web application (see the demo applications) requires a separate build.xml file (in DDS/applications/${application}/bin), including the following targets:

- create-application
- delete-application
- load-application-data
- load-data-sets
- build
- run
- create-war

Ant targets are described below in alphabetical order, including parameters and an indication of their primary purpose, which can be for administrator use or for developer use. See the file DDS/bin/build.xml and the demo applications for specifics.

**create-database**

Creates a database in the XML Store (Administrator).

Parameters:

- xdb.bootstrap
- superuser.password
- name
- admin.password

**create-database-layout**

Creates the database's main library structure, as required by DDS (Administrator).

Parameters: none

**delete-dataset**

Deletes a data set from the database (Administrator).

This implies that the library with the dataset's name is removed from the /DATA library in the database (including everything under it).

Parameters:

- dataset

**create-application**

Creates an application in the database (Administrator).

This implies that a library is created in the database with the name ${application}. The library's further properties are (unless overridden) taken from bin/ant-xdb.properties. Actually, this target just invokes the "create-application" target in the application's own build.xml (to be found in applications/${application}/bin).

Parameters:

- application

**delete-application**

Deletes an application from the database (Administrator).

This implies that the library with the application's name is removed from the /APPLICATIONS library in the database (including everything under it). Actually, this target just invokes the "create-application" target in the application's own build.xml (to be found in applications/${application}/bin).

Parameters:
- application

**load-application-data**

Loads an application's data (e.g. xforms, xprocs, xslt files, etc.) into the database (Administrator).

Actually, this target just invokes the "load-application-data" target in the application's own build.xml (to be found in applications/${application}/bin).

Parameters:
- application

**load-data-sets**

Loads an application's data sets into the database (Administrator).

This target invokes the "load-data-sets" target in the application's own build.xml (to be found in applications/${application}/bin).

Parameters:
- application

**load-all-data**

Loads both an application's data and its data sets into the database (Administrator).

Parameters:
- application

**clean**

Deletes everything in the build folder, and cleans op anything left behind by Eclipse or by the GWT shell and compiler (Developer).

Building an application (using the “build” target), and running an application in hosted more (using the “run” target), creates folders and files in the DDS/build folder. The clean target will clean all of those up for you.

Parameters: none

**create-template**

Creates a template folder structure under the applications folder for the development of a new DDS application (Developer).

The root of the application structure will be applications/${application}. Also, a build.xml and build.properties template are created in applications/${application}/bin.

The complete folder structure of DDS/applications/${application} is:
- bin(with, among others, build.xml and build.properties)
- data(for data sets: xml and blob content)
    - application(for application resources that go into the database: xforms, xprocs, xslt)
        - configuration
        - resources
            - xforms
            - xproc
            - xslt
- lib(for application-specific jars)
- resources(for application resources that need to be on the application’s classpath)
- src(for the application’s Java (including GWT) and other sources)
- war

- WEB-INF (e.g. for the application’s web.xml file)

Use only what is needed. For example, if the application is not a servlet based application, or if the default web.xml is fine, you can omit the war folder.

Parameters:
- application

**build**

Prepares an application by compiling all source code, both client and server (Developer).

The result is placed in build/${application}. Actually, this target just invokes the "build" target in the application's own build.xml (to be found in applications/${application}/bin).

By default, build will call “-build-client” and “-build-server” in DDS/bin/build.xml.

- build-client is tailored for GWT applications: it invokes the GWT compiler (with the application’s jars from its lib folder on the classpath), on the generic DDS-GWT code and the application’s GWT code (under src) and put the resulting files in DDS/build/{application}
- build-server will compile all server code (under src) of the application (with the application’s jars from its lib folder on the classpath), and put the class files in DDS/build/{application}

Parameters:
- application

**run**

Runs an application in "hosted" mode (Developer).

Actually, this target just invokes the "run" target in the application's own build.xml (to be found in applications/${application}/bin).

Hosted mode runs the entire web-application in a GWT-provided container, which hosts the application (including the client) in Java. This is especially useful for debugging GWT applications.

The application’s own configuration files are taken into account:

- application-bootstrap.xml
- web.xml

Parameters:
- application

**create-war**

Creates a .war file for an application, for deployment under Tomcat (Developer).

Actually, this target just invokes the "create-war" target in the application's own build.xml (to be found in applications/${application}/bin).

All the application’s own configuration files are included in the war:

- application-bootstrap.xml
- web.xml

Parameters:
- application

**run-xforms-builder**

Runs the XForms Builder (Developer).

Make sure EMC Documentum Forms Builder 6.5 SP1 is installed and properly linked to from DDS/bin/ant-environment.properties

Parameters: none

# DDS Ant tasks

DDS includes a set of Ant tasks that are used by DDS Ant targets. Administrators and developers can use these Ant tasks for fine-tuning tasks, and for creating additional Ant targets:

- create-dataset
- import-application-data
- import-data
- security-tool

**create-dataset**

Creates a dataset in the database under the DATA library.

Attributes:

- dataset: the name of the dataset (required)
- databaseref: a reference to the database construct (required)
- overwrite: indicator that data that already exists in the database should be overwritten.

  If true, the data set library is replaced by a new library. Default = false.

  **Note:** If a dataset with the same name already exists and overwrite = true, the existing dataset will be lost.
- quiet: determines what information will be shown on the console.

  If false, all import information will be shown. If true, only warnings and errors are shown. Default = false.
- localeaware: indicator that the dataset is aware of locales. If false, no locales can be used. Default=false

Nested elements:

- libraryoptions: sets library options of the new dataset library.

  Nested element: option, with attributes name and value.

  Available options:
  - concurrent-library (default = true)
  - concurrent-namebase (default = true)
  - lock-with-parent
  - documents-do-not-lock-with-parent

```
<dds:create-dataset
  dataset="${application}"
  databaseref="demo.database"
  localeaware="true"
  overwrite="false"
  quiet="false">
  <libraryoptions>
    <option name="concurrent-library" value=“true"/>
    <option name="concurrent-namebase" value=“true"/>
  </libraryoptions>
<dds:create-dataset>
```

**import-application-data**

Imports application data like XForms, XProcs and stylesheets.

Attributes:

- application: the application name (required)
- databaseref: a reference to the database construct (required)
- overwrite: indicator that existing data in the database can be overwritten

  If true, data that already exists in the database will be replaced by corresponding imported data. Default = false.

- quiet: determines what information will be shown on the console.

  If false, all import information will be shown. If true, only warnings and errors are shown. Default = false.

Nested elements:

- libraryoptions: During import, missing libraries are created automatically. This element sets library options for these new libraries.

  Nested elements: option, with attributes name and value.

  Available options:
  - concurrent-library (default = true)
  - concurrent-namebase (default = true)
  - lock-with-parent
  - documents-do-not-lock-with-parent

- xmlfileset : Extension of Ant fileset, to specify all files that must be imported as XML. See Ant fileset attributes.

  Nested elements: domconfiguration, to set xml parse options, nested element parameter with attributes name and value.

  Available options and default values are specified in the W3C DOM Load and Save spec and the database manual.

- blobfileset : Extension of Ant fileset, to specify all files that must be imported as blob. See Ant fileset attributes.

**Note:** The content filesets are locale aware: xmlfileset and blobfileset have an additional, optional attribute locale. See the import-data task for an example.

```
<dds:import-application-data
    application="${application}"
    databaseref="demo.database"
    overwrite="true"
    quiet="false">

    <libraryoptions>
      <option name="concurrent-library" value=“true"/>
      <option name="concurrent-namebase" value=“true"/>
    </libraryoptions>

    <dds:xmlfileset dir="${dds.garage.data.application}">
                  <domconfiguration>
        <parameter name="validate" value="true"/>
      </domconfiguration>
      <include name="**/*.xpl"/>
      <include name="**/*.xsl"/>
    </dds:xmlfileset>

    <dds:blobfileset dir="${dds.garage.data.application}">
      <include name="**/*.txt"/>
      <include name="**/*.properties"/>
      <include name="**/*.css"/>
    </dds:blobfileset>
  </dds:import-application-data>
```

**import-data**

Imports content data and schemas.

This task has the same attributes and elements as import-application-data. In addition, import-data can also contain the following additional elements:

- schemas: use to specify schemas. The schemas are stored in the catalog of the database.

  Nested elements: schema, with attributes:

  - schematype: specifies the type of the schema (required)
    Available schema type values:
    - XMLSchema (see http://www.w3.org/2001/XMLSchema) or
    - DTD (see http://www.w3.org/TR/REC-xml)
  - Systemid: the system Id of the schema during import (required)
  - publicid: The public Id used in the XML data to reference the DTD (required if schema is a DTD)

- metadatafileset : Extension of Ant fileset, to specify all metadata files that must be imported. See Ant fileset attributes.
  Optional additional attributes:
  - metadataextension: specifies a file extension. This extension is removed during import.

  **Note:** The content filesets are locale aware: xmlfileset, blobfileset and metadatafileset have an additional, optional attribute locale.
  **Note:** Do not set locale if your dataset is not locale aware

```
<dds:import-data dataset="${dataset.name}"
  databaseref="demo.database">

      <dds:schemas>
        <dds:schema schematype="http://www.w3.org/TR/REC-xml"
                systemid="${dds.garage.data}/dtd/topic.dtd"
                publicid="-//OASIS//DTD DITA Topic//EN"/>
        <dds:schema schematype="http://www.w3.org/2001/XMLSchema"
                systemid="${dds.garage.data}/xsd/task.xsd"/>
       </dds:schemas>

      <dds:xmlfileset
       dir="${dds.garage.data.collection.repository.britannica}/eng"
       locale="eng">
        <include name="**/*.xml"/>
        <include name="**/*.ditamap"/>
      </dds:xmlfileset>
      <dds:blobfileset
       dir="${dds.garage.data.collection.repository.britannica}/eng"
       locale="eng">
        <include name="**/*.png" />
      </dds:blobfileset>

      <dds:metadatafileset
       dir="${dds.garage.data.collection.repository.britannica}/eng"
       locale="eng" metadataextension="metadata">
        <include name="**/*.metadata"/>
      </dds:metadatafileset>

    </dds:import-data>
```

**security-tool**

Tool for public key cryptography.

The tool has two functions:

1. Generate keys

2. Encrypt a password

Attributes:

- command: either encrypt or generate (required).
- privateKeyPath: the path to the private key. Default value: "./DDSPrivateKey.dat".
- publicKeyPath: the path to the public key. Default value: "./DDSPublicKey.dat".
- password: the password to encrypt. (required if command = encrypt)
- var: the result of password encryption is stored in this attribute. Use it as: var="password". (required if command = encrypt)

```
<!-- encrypt password -->
<dds:security-tool
   command="encrypt"
   privateKeyPath="${application.bin.dir}/DDSPrivateKey.dat"
   publicKeyPath="${application.bin.dir}/DDSPublicKey.dat"
   password="secret"
   var="password"/>
<echo message="${password}"/>

<!-- create new keys -->
<dds:security-tool
   command="generate"
   privateKeyPath="${application.bin.dir}/DDSPrivateKey.dat"
   publicKeyPath="${application.bin.dir}/DDSPublicKey.dat"/>
```

# API, Frameworks and Services

## Core concepts

This chapter introduces core concepts, gives an overview of the development process, and describes the DDS Services, Frameworks, and some other amenities.

**Server and client**

A DDS Application can typically be seen as having 2 major components:

- a server, which provides back-end functionality through Services
- a client, which uses the Services to provide functionality to the end user.

The client typically offers a user interface, and DDS provides facilities to build user interfaces, which are integrated with the functions offered by the Services.

The server typically manages data storage and retrieval, and provides (among other things) an abstract layer which allows access to data in a transparent, storage-independent way.

The Services on the server can be used by both client and server components. To the client they can be exposed using various protocols, typically through a Client Service. A Client Service therefore wraps a Server Service (hereafter simply referred to as Service) and makes the functionality of the Service available over a particular protocol.

In addition to the provided Services, it is also possible for developers to build Services of their own.

Client Services have been provided for GWT as GWT Services , a wrapper that enables GWT client components to use the DDS Services.

**Application & User**

Central to the server of an Application is the Application Framework, which contains the Application object. The Application object represents the server component, and offers access to the server resources and facilities.

The Application is configured using the `application-bootstrap.xml` document, which contains the minimum of information needed to start up the Application, including references to other configuration files needed by the Application to set up its resources and facilities.

End users of the Application are represented by the User concept. This concept models an end user of the DDS Application, as opposed to, for example, the user concept of a database that is accessed by the Application.

See Application Framework for more information.

**Data Access**

An important function of an Application is to provide access to data, which can be in various facilities: databases, file systems etc. The Persistence Layer transparently models such storage facilities by means of the Store concept, which can represent a database, a file system, a Documentum Store, etc.

Locations and Containers represent the equivalents of folders and files – that is, of a hierarchical organization of places where documents can be stored, and of the documents themselves.

For XML data, the XML Node concept provides access to individual nodes in the XML documents.

Any actions on the data (such as creating Locations, retrieving or storing data from or into Containers, deleting data) happen through Persistors which implement the correct behaviour for the Stores.

A Store is always accessed through a Session, which (provided the Store supports it) offers transactional behaviour. One Store is singled out and configured as the Main Store of the Application. The Main Store is where all configuration data and application-specific resources will be kept.

The Application keeps track of and offers access to the Stores that have been configured by means of the StoreManager. Configuration of the available Stores happens through the `Stores.xml` document in the Main Store.

See Application Framework and Persistence Layer for more information.

**Data Organization**

Whereas the Persistence Layer offers lower-level access to the storage facilities, the Structures Framework offers higher-level concepts for keeping data organized.

A DataSet is a set of data which logically belongs together, and is independent of the Application. It is up to the Developer and Administrator to determine how data should be divided into DataSets. This will depend on how the data should be used. A DataSet is always stored inside a single Store - it is not possible to store half of a DataSet in one Store, and the other half in another Store.

Access to DataSets (for reading or writing data) happens through the Alias of the DataSet, which is unique in an Application. This means there is no need for the Developer to know in which Store, or at what location in the Store the DataSet resides – this is resolved transparently.

The Application keeps track of the DataSets that have been configured for it. An Application can have access to multiple DataSets, and conversely, a DataSet can be accessed by multiple Applications.

For internationalization (I18N) purposes, DDS provides the concept of a Locale, which is a subset of a DataSet, and contains the data for a particular language, country or a combination thereof.

DataSets and Locales are examples of a Structure, which is the generalized concept representing any set of Locations and Containers together with a Strategy (called StructureStrategy) which determines the physical location where the data will be stored.

The Application keeps track of and offers access to the DataSets and Locales that have been configured by means of the StructureManager. Configuration of the DDS DataSets and DDS Locales which are available happens through the `Structures.xml` document in the Main Store. New DDS DataSets and DDS Locales can also be created with the StructureManager.

See Application Framework and Structures Framework for more information.

**Services**

The Application manages the Services offered by the server component through the Service Framework.

This framework offers a Service concept which models a Service as having a lifecycle: a Service can be started and stopped, as well as paused and resumed.

The (non-administrative) functions offered by the Service are only available while the Service is running. In any other state, only the administrative functions of the Service are usable. These are functions that typically are only available to an DDS Administrator.

The functions offered by a Service are defined by the API defined in a Java interface, and access to a function of a Service is by calling the appropriate Java method. Client components can access the Services indirectly through a Client Service which acts as an intermediary between the Client and the Service.

The following Services have been provided :

- UserService: functionality related to Users
- XProcService: allows the execution of XProc pipelines
- LogicEngineService: access to the Logic Engine functionality
- TokenService: functionality for mapping Session Tokens from client sessions to Users
- ResponseService: functionality related to processing and storing input received from Users

The Application keeps track of, and offers access to, the Services that have been configured by means of the ServiceManager. Configuration of the Services which are available happens through the `Services.xml` document in the Main Store.

The Application is itself modelled as a Service, which means the Application can be started, paused and resumed and stopped as well. The Application will automatically initialize and start all the configured Services when it starts up, pause all Services when it is itself paused, and so on.

See Application Framework and Services Framework for more information.

**Operations**

The Application also offers access to the Operation Framework, which underlies the Services. It models basic functions as Operations, which can be executed independently.

Operations can also be strung together in an OperationSequence, which can execute all the Operations in one go, as if they were a single Operation. To a limited extent, an OperationSequence can have transactional behaviour.

An Operation library has been provided, modelling all persistence actions. The Operation framework takes care of Sessions, rollback and concurrence, and is the preferred way of accessing and storing data. Operations are executed using the OperationManager.

See Application Framework and Operation Framework for more information.

# Server-side Application Development

Application development usually starts with the server side of the Application.

1. Prepare data.

   Upload the DataSets the Application will use.

   You can use Ant tasks for this purpose.

2. Create the Application.

   Some basic setup information must be present in the Main Store (the Store that is the “home” of the Application).

   Ant tasks are provided for creating the minimal layout. Alternatively, you can use the Admin application to create the layout.

3. Configure additional Stores, if any Stores other than the Main Store need to be configured.

   Configure any and all Stores that the Application needs access to, except for the Main Store (which is configured in the application-bootstrap.xml file).

   You must manually create a `Stores.xml` configuration file, which contains the configuration data for the Stores. This file must be stored in the Main Store, by default in `/APPLICATIONS/<application_name>/Stores.xml`. Every Store should have its own unique Alias for the Application, which the Application must use throughout to refer to that Store.

   When the Application has been started, all the Stores configured in the document will be available in the StoreManager.

   See Application Configuration for more information.

4. Configure Structures, if required.

   If no Structures (DataSets that the Application requires access to) need to be configured, this step can be skipped.

   You must manually create a `Structures.xml` configuration file, which contains the configuration data for the DataSets. This file must be stored in the Main Store, by default in `/APPLICATIONS/<application_name>/Structures.xml`. Each DataSet has to be assigned a unique Alias, for use throughout the Application to refer to the DataSet.

   The reason for the Alias is that, while the Id of the DataSet will be unique in a Store, it is still possible that a DataSet with the same name resides in another Store which will also be accessed by the Application.

   When the Application has been started, all the Structures configured in the document will be available in the StoreManager.

   See Application Configuration for more information.

5. Configure Services, if required.

   If no Services will be available in the Application, this step can be skipped.

   You must manually create a `Services.xml` configuration file, which contains the configuration data for the Services. This file must be stored in the Main Store, by default in `/APPLICATIONS/<application_name>/Services.xml`.

   All the Services configured in the file will be available in the ServiceManager, when the Application has been started. They are all started automatically during Application startup.

   When the Application has been started, all the Services configured in the document will be available in the StoreManager.

   See Application Configuration for more information.

6. Start **Tomcat** (using a separate command-line session).

   To access the Main Store, the Application needs to be configured with a StoreUser (meaning the user at Store-level, not a DDS User !), which will be used to connect to the Store. The

password required by the Application will be stored, in encrypted form, in the Application Bootstrap.

The SecurityTool Ant task can be used to generate new private and public keys for encryption, instead of using the included default keys. The tool can also be used to enter a password. The encrypted version will then be shown, and can be copied for use in the Application Bootstrap (see below).

7. Configure the Application Bootstrap.

   The application-bootstrap.xml file contains the information needed to instantiate and start the Application. The following should be supplied:

   - the application name
   - configuration for the Main Store, including a Default Store User which has administrative rights in the Store
   - the paths to the public and private keys on the file system
   - the paths to the Stores.xml, Structures.xml and Services.xml configuration files in the Main Store
   - a Default Structure Strategy, specifying the layout in the database

   For details on the format of the XML, see Application Configuration .

8. Deploy and start the server component of the Application.

   It is possible to develop the client component at this point, if all the server processing needs are covered by the standard amenities.

# Adding Server Functionality

The following mechanisms are available for adding server functionality:

- Operations can be used to contain and expose functions that are fairly low-level, self-contained, and that should be available in a variety of contexts in the server component.
- Services can be used to expose higher-level functions.

Typically, Service implementations will use Operations.

**Creating New Operations**

Implementation of a new Operation requires two classes, one implementing the Operation interface, and one implementing the OperationExecutable interface.

Operation contains all the information needed to carry out the coded task. Typically the information is provided in the constructor, and getters are provided so the OperationExecutable can access the information.

OperationExecutable contains the actual implementation. When the Operation is executed, it is instantiated by the Operation Framework, and the Operation is provided so the OperationExecutable can access the information it needs.

If the Operation accesses any Stores, the Stores should be declared using the declareStore() method. When the Operation is executed, the OperationExecutable will receive a Map containing Sessions for all the involved Stores. These Sessions can be used to perform persistence actions on those Stores.

Some abstract, partial implementations are provided in the `com.emc.documentum.xml.dds.operation.library.basic` package, which should be extended when creating Operations.

See Application Framework and Persistence Layer for more information.

**Creating new Services**

To implement a Service, first create an interface containing the API for the new Service. This interface should extend the `com.emc.documentum.xml.dds.service.Service` interface.

The actual implementation should extend the `com.emc.documentum.xml.dds.service.impl.ServiceImpl` class, and implement the interface created for the new Service. This class provides the lifecycle implementation.

Some abstract methods provide hooks to implement behaviour when the state of the Service is changed. For example, data structures should be initialized by implementing the executeInitialization() method. The Service functions that provide normal runtime functionality should throw a ServiceNotAvailableException if the Service is not in the RUNNING state when called.

# Persistence Layer

The Persistence Layer provides abstractions which allow the Developer to persist, retrieve and manage data without reference to the type of the underlying storage facilities.

**Concepts**

The Persistence Layer is built around the following major storage concepts:

An Implementation provides Persistence Layer support for specific types of storage facilities.

A Store models the underlying storage facility. A Store may correspond to a file system, an XDB database (but not a federation), or any storage facility that is organized hierarchically.

A StoreChild is the parent concept of the following concepts, which model entities contained in a Store:

- A Location models a location in the storage facility where data can be stored. Locations are organized hierarchically, with a single root Location. A Location may correspond to a directory or folder on a file system, or a library in an XDB database.
- A Container models the storage object that contains the actual data. A Container may correspond to a file on a file system, or a document or blob in an XDB database.

- An XMLNode models any XML Node in the Store. This includes Document nodes (as per DOM) as well as nodes inside a Document.

  The basic model consists of a Container and an XPointer expression, where the referenced Node is a Node inside the Container, and obtained by resolving the XPointer expression. However, the XMLNode can actually contain an `org.w3c.dom.Node` object, but only in case the XML Node was returned by a Persistence operation (such as getNode() or getChildren() in the XML Persistor). This result should almost never be used directly, except as input to another Persistence Operation. In all other cases it should be transformed to another representation, such as a Serialized Node.

  **Note:** Depending on the type of Store, other entities may be treatable as XML Nodes. Such behaviour should NOT be relied upon when writing code that should be portable across different types of Store. For example, in XDB the Libraries can be treated as XML Nodes, and support is provided for this in the XDB implementation of the Persistence concepts.

A StoreUser models the user concept at the level of the Store, for example an operating system user on a file system, or an XDB user in an XDB database.

A ContentDescriptor is used when storing or retrieving data, to specify to the Persistor what the content type of the data is.

The Data concept models data retrieved from the Store.

The Metadata concept models the metadata which can be associated with a Container. Metadata can be stored in several ways, which is modeled by Metadata Schemes:

- XDB Metadata consist of key-value pairs, which are stored in the XDB database using a specific mechanism. It can only be used for Containers in an XDB Store.
- Documentum Metadata consist of a separate XML document which is stored in a particular location in a DataSet. It can only be used for Containers in DataSets which have the DocumentumStructureStrategy. The type of the Store is irrelevant.

A Session models a transaction on the Store, if transactional behaviour is available on the Store. A Session is always created for a particular StoreUser on a particular Store, and must be provided to the Persistor in order to perform a Persistor action.

The following concepts model options that can be set on a Location or a Container, and that are specific to the type of Store:

- LocationOptions can model such things as file system permissions on file systems, or locking options in an XDB database
- ContainerOptions can model such things as file system permissions on file systems, or locking options in an XDB database

The following concepts provide ways of interacting with the storage facility:

A Persistor is an object which provides actions that can be performed on Containers and Locations in a Store, including:

- Checking the existence of Locations, Containers and StoreUsers
- Creating Locations, Containers and StoreUsers
- Deleting Locations, Containers and StoreUsers
- Persisting content into Containers
- Retrieving content from Containers
- Associating Metadata with Containers
- Retrieving Metadata associated with Containers

An XMLPersistor is an object which provides actions that can be performed on XMLNodes a Store, including:

- Retrieving the number of children of an XMLNode

- Retrieving XMLNodes or their children
- Inserting XML fragments
- Moving, copying and removing XMLNodes
- Retrieving and setting attributes on XMLNodes

**Interfaces**

The Persistence Layer concepts have been codified into interfaces in the `com.emc.documentum.xml.dds.persistence` package.

**Store**

The Store interface (`com.emc.documentum.xml.dds.persistence.Store`) provides accessors to the following Store properties:

- Store Alias: the Application-specific identifier assigned to a Store. It is configured explicitly in the appropriate configuration file, and has to be unique across Stores for the Application.
- Store Id: the id of the Store. The Store Name models information necessary for the implementation to connect to the correct storage facility; it is not used as a means of identifying Stores to the Application.

  The exact semantics depend on the type of Store and the implementation. For example, the Store Id for an XDB database is the name of the database. In a Windows file system environment, the Store Id may be the name of a disk (C:, D:, ...). In other environments it may not be relevant.

- Store Type: an Enumeration constant specifying the type of the storage facility represented by the Store, for example file system, or XDB database.
- Separator: represents the character used to construct paths in the Store. For example, in an XDB database or on a Linux file system, this is the forward slash character. On a Windows file system it is a backslash character.
- Default StoreUser: the default StoreUser for persistence actions on the Store, if no StoreUser has been specified explicitly (see SessionStoreUserStrategy). It can be configured explicitly in the appropriate configuration file.
- XMLPersistor: a Store-specific implementation of the Persistor for XML-related persistence operations, suitable for this Store.
- XQueryExecutor: a Store-specific implementation of the XQueryExecutor object, suitable for this Store.

The Store interface provides the following methods:

- getSession(StoreUser storeUser): creates a new Session for the specified StoreUser for the Store, which can be used for persistence actions.
- getLocation(): creates a new Location object which is valid in the specified Store, with the specified path.
- getContainer(): creates a new Container object which is valid in the specified Store, with the specified Location as parent and with the specified Container Name.

**StoreChild**

The StoreChild interface (`com.emc.documentum.xml.dds.persistence.StoreChild`) provides accessors to the following StoreChild properties:

- Name: the name of the StoreChild.

  For Locations, this might be the directory name in a file system, or the library name in an XDB database. In that case it is equal to the last component of the Location Path. For Containers, this might be the file name in a file system, or the document name in an XDB database. For XMLNodes, the name depends on the type of the XML Node, and could be the element name, the attribute name, etc.

- Store Type: the type of Store for which this is a valid Location.

- Store: represents the Store the StoreChild belongs to.
- Store Alias: represents the alias of the Store the StoreChild belongs to.
- Path: the full path of the StoreChild in the Store. Generally represented as the Path Components joined together by the Separator character for the Store.

The StoreChild interface provides the following methods:

- isLocation(): indicates whether the StoreChild is a Location.
- isContainer(): indicates whether the StoreChild is a Container.
- isXMLNode(): indicates whether the StoreChild is an XMLNode.
- asNode(): interprets the StoreChild as an XML Node, if possible. This will not work for Stores where Locations cannot be treated like XML Nodes.

**Location**

The Location interface (com.emc.documentum.xml.dds.persistence.Location) provides accessors to the following Location properties:

- Parent: the parent Location.

  The Parent of the Root Location of a Store is *null*.

- Path Components: the individual components making up the Path.
- Root: indicates whether the Location represents the Root Location for the Store.
- Options: the LocationOptions for the Location. If null, the default DDS LocationOptions will be used for persistence actions when needed.

The Location interface provides the following Location methods:

- getChildLocation(String name): creates a new child Location object with the specified name.
- getChildContainer(String name): creates a new child Container object with the specified name.
- getDescendantLocation(String relativePath): creates a descendant Location, in a Path relative to the ancestor Location specified by the relativePath parameter.
- getDescendantContainer(String relativePath, String name): creates a descendant Container, in a Path relative to the ancestor Location specified by the relativePath parameter, with the specified name.
- deepCopy(): creates a new Location object representing the same Location.

**Container**

The Container interface (com.emc.documentum.xml.dds.persistence.Container) provides accessors to the following Container properties:

- Container Type: the type of the Container. This is a Store-specific Enumeration constant. For example, on the file system there is only the “file” container type, whereas in an XDB database there are two types: Document and Blob.
- Location: the parent Location in which the Container resides.
- Path: the full path of the Container in the Store.
- Options: the ContainerOptions for the Container.

The Container interface provides the following method:

- getXMLNode(String xpointer): creates a new child XMLNode, pointing to the XML Node specified by the Xpointer expression.

**XMLNode**

The XMLNode interface (com.emc.documentum.xml.dds.persistence.XMLNode) provides accessors to the following XMLNode properties:

- Container: the Container to which the Node belongs.
- XPointer: the XPointer expression pointing the XML Node in the Container.

The XMLNode interface provides the following method:

- asNode(): returns the org.w3c.dom Node contained in the XML Node, if it is contained. This only returns the DOM Node if the XMLNode is the result of a Persistence Operation. The Node may not be valid outside the context of a transaction (this is the case in XDB), so this

method should only be used for the specific purpose of accessing DOM Nodes in the context of a transaction.

**ContentDescriptor**
The ContentDescriptor interface (com.emc.documentum.xml.dds.persistence.ContentDescriptor) provides accessors to the following ContentDescriptor property:

- XML: a boolean indicating whether the content is XML or not.

**Data**
The Data interface (com.emc.documentum.xml.dds.persistence.Data) provides accessors to the following Data properties:

- Content: contains the actual wrapped data.
- MIME Type: indicates the type of the content:
    - application/octet-stream for binary content
    - application/xml for XML content, and
    - application/java-serialized-object for serialized objects.

**Metadata**
The Metadata interface (com.emc.documentum.xml.dds.persistence.Metadata) provides access to the following Metadata property :

- Scheme: the MetadataScheme of the object

**StoreUser**
The StoreUser interface (com.emc.documentum.xml.dds.persistence.StoreUser) provides accessors to the following StoreUser properties:

- User Id: the user Id of the StoreUser, used for authentication with the Store.
- User Password: the password of the StoreUser, used for authentication with the Store.
- Administrator: indicates whether the StoreUser is an administrator in the Store.
- Store Type: the type of Store for which this is a valid StoreUser.
- Store Alias: the Store for which this is a valid StoreUser.

**Persistor**
The Persistor interface (com.emc.documentum.xml.dds.persistence.Persistor) provides accessors to the following Persistor properties:

- Store Type: the type of Store for which this is a valid Persistor.
- Serializer: the Serializer object (see Serialization Framework) which will be used to serialize and deserialized Java objects when they are persisted into or retrieved from the Store by the Persistor.

The Persistor interface provides the following methods for performing persistence actions on a Store:

- exists() methods: check whether the specified Locations or Containers exist in the Store, and return the result as a boolean.
- create() methods: create the specified Location or Container in the Store.

  The createPath parameter specifies whether or not to create the parent Locations, if they do not exist. The replace parameter specifies what should happen if the Container already exists : if true, it will be replaced by the newly created Container, otherwise an exception will be thrown.

  The ContainerOptions and LocationOptions set on the Container and Location parameters will be applied.

- delete(): deletes the specified Location or Container in the Store.
- listChildren(): lists the child Locations and Containers for the specified Location. Boolean parameters are included to specify whether Locations and/or Containers should be listed, and whether descendants should be listed.
- persist(): stores the specified Data object into the specified Container.
- retrieve(): retrieves the XML from a Container and return it wrapped in a Data object.

- getOptions(): retrieves the LocationOptions for the specified Location.
- setOptions(): sets the LocationOptions for the specified Location.
- getMetadata(): retrieves the Metadata for the specified Container and Scheme
- setMetadata(): sets the Metadata for the specified Container and Scheme
- existsUser(): checks whether the specified StoreUser exists in the Store.
- createUser(): creates the specified StoreUser in the Store.
- deleteUser(): deletes the specified StoreUser from the Store.

**XMLPersistor**
The XMLPersistor interface (com.emc.documentum.xml.dds.persistence.XMLPersistor) provides accessors to the following XMLPersistor property:

- Store Type: the type of Store for which this is a valid Persistor.

The XMLPersistor interface provides the following methods for performing XML persistence actions on a Store:

- getChildCount() methods: returns the number of children of the specified XMLNode.
- getChildren() methods: retrieves the children of an XML Node, optionally filtered by node type.
- getChildrenByRange(): retrieves a subset of the children of an XML Node, specified by a start and end index.
- getNode(): retrieves the specified Node.
- getNodes(): retrieves a list of Nodes.
- insert(): inserts an XML fragment into or before the specified node.
- copy() and move() methods: copy or move the specified XML Node into or before the specified target XML Node.
- remove(): removes the specified Node.
- setAttribute() and setAttributes() methods: set the attributes on an XML Node.

**Session**
The Session interface (com.emc.documentum.xml.dds.persistence.Session) provides accessors to the following Session properties:

- Store Type: the type of Store for which this is a valid Session
- Store: the Store for which the Session was created.
- User: the StoreUser for which the Session was created.
- Session: the actual store-specific Session object. For example, for an XDB database, this is an XhiveSessionIf object.

The Session interface provides the following methods abstracting transactional behaviour:

- begin(): starts the transaction.
- commit(): commits the transaction.
- rollback(): rolls back the transaction

**LocationOptions**
The LocationOptions interface (com.emc.documentum.xml.dds.persistence.LocationOptions) provides accessors to the following LocationOptions property:

- Store Type: the type of Store for which these are valid LocationOptions.

**ContainerOptions**
The ContainerOptions interface (com.emc.documentum.xml.dds.persistence.ContainerOptions) provides accessors to the following ContainerOptions property:

- Container Type: the type of Container for which these are valid ContainerOptions.

**Usage**

The recommended (and easiest) way to work with the Persistence Layer is by using the Operation Framework, which encapsulates all the persistence operations, and takes care of handling Sessions, StoreUsers etc. Directly using the Persistence Layer should only be considered for special cases, for

example when you need to create Sessions and call the Persistor directly, notably for fine control over transactions.

**Note:** An important aspect of working with the Persistence Layer is that, not unlike the java.io.File class, there is no close coupling between the Store, Location, Container and XMLNode objects on one hand and the actual storage facility on the other hand. This loose coupling means that creating Store, Container, Location or XMLNode objects, or changing them (including setting their Options) does not affect the storage facility at all, and vice versa. Only the “action methods” of Persistor or XMLPersistor can actually affect the storage facility, or retrieve information about it.
**Note:** Persistors, Locations or Containers for different types of Store are not “compatible” with each other. For example, LocationOptions for different types of Store will often be incompatible. If you construct a Location on an XDB Store using its getLocation() method, you will not be able to use that object with the create() method from a Persistor on a file system Store: such a call will throw an InvalidStoreException.

# Using the Persistence Layer

The Persistence Layer allows the Developer to persist, retrieve and manage its data if finer control over transactions is needed than what is provided by the corresponding Operations.

**Working with Sessions**
**Note:** The Operation framework manages Sessions transparently. This section is only relevant if the corresponding Operations cannot be used, and fine control is needed over transactions.

Any persistence action requires a Store and a StoreUser to create a Session on the Store, using the Store getSession(StoreUser) method.

This produces a Session which encapsulates an actual transaction on the Store. It can only be used for persistence operations on that particular Store.

In order to start the Session, and before using it in the persistence transactions, it must be activated by calling its begin() method.

When all persistence actions have been performed, they should be committed by calling the commit() method on the Session, which will actually commit the changes. If for some reason the operations need to be rolled back, the rollback() method can be called instead.

**Check whether the library “/MyApp/Documents” exists in the Store**

```
Store store = Application.getStore(“MyStore”);
StoreUser storeUser = store.getDefaultStoreUser();
Location location = store.getLocation(“/MyApp/Documents”);
Persistor<Object> Persistor = PersistorFactory.constructPersistor(store);
Session session = store.getSession(storeUser);
session.begin();
persistor.create(session, location);
session.commit();
```

**Working with Locations**

The easiest way to obtain a Location object is to call the getLocation() method on a Store directly. This creates a Location object of the correct type, with default LocationOptions for that Store.

**Constructing a Location**

```
Location location = store.getLocation(“/MyApp/Documents”);
```

The Store property of the Location will be set to the Store on which the Location was constructed.

For purposes of navigation, the getParent(), getChildContainer(), getChildLocation(), getDescendantContainer() and getDescendantLocation() methods available in the Location provide convenient ways to create new Location and Container objects.

**Note:** These methods do not in any way affect or take into account the state of the actual storage facility.

Calling:

```
newLocation = myLocation.getDescendantLocation(“foo/bar”);
```

returns a Location object two levels down in the Store hierarchy, which may or may not exist in the actual storage facility.

**Note:** getDescendantContainer() and getDescendantLocation() are only applicable to actual, true descendants ! Moving sideways or up in the Location structure is not possible with this method.

For instance, in the XDB structure, calling

```
newLocation = myLocation.getDescendantLocation(“../foo/bar”);
```

would not yield a “nephew” Location (the child “bar” of the Location “foo” which has the same parent Location as myLocation) but a descendant three levels down. The grandparent of the newLocation would be the “..” library (which is a valid name for an XDB library!), which would itself be the child Location of myLocation.

**Applying LocationOptions**

If the LocationOptions are set on a Location, they will be applied when the Location is created. If they are null for the Location, the default LocationOptions will be used.

When Locations are being created implicitly (if the createPath parameter is set to true for persistence actions), the LocationOptions of the Location (provided as a parameter or implicitly present in the Container object) will be used for all the newly created Locations.

Setting the LocationOptions on existing Locations may not always be possible. With XDB, for instance, the library options can only be applied when the library is created. The Persistor setOptions() method will always throw an Exception.

**Working with Containers**

The easiest way to obtain a Container object is to call the getContainer() method on a Store directly. This creates a Container object of the correct type, with default ContainerOptions for that Store.

```
Container container = store.getContainer(myLocation, “Document.xml”,
ContentType.XML);
```

or:

```
Container container = store.getContainer(“/foo/bar/”, “Document.xml”,
ContentType.XML);
```

The Location property of the Container will be set to the Location in which the Container was constructed.

**Working with XMLNodes**

The easiest way to obtain an XMLNode object is to call the getXMLNode() method on a Container. This creates an XMLNode object of the correct type, using the provided Xpointer expression. An Xpointer expression is always of the form “xpointer(…)”.

```
XMLNode xmlNode = container.getXMLNode(“xpointer(/library/book/title)”);
```

# Persistence Layer and XDB

DDS includes an xDB implementation, in the `com.emc.documentum.xml.dds.persistence.xdb` package.

The following objects provide extra functionality on top of what is offered by the standard interfaces:

**Session**
The XDBStore object has some additional properties, with corresponding accessors:

- Bootstrap: the URI of the xDB bootstrap, needed to connect to the xDB database.
- Cache Pages: specifies how much memory will be allocated to the xDB database driver when it is instantiated. See the xDB documentation for details.

The XDBStore object provides the following methods:

- getSession(): returns an XhiveSessionIf object.
- connect(): initializes the driver and connects the Store object to the actual database.
- disconnect(): closes the driver.

**XDBLibraryOptions**

The XDBLibraryOptions model the options that can be set on an XDB library Location.

**Note:** These options can only be specified at library creation time. The setOptions() method from the XDBStandardPersistor will always throw an Exception.

The following options can be set for XDB libraries through the XDBLibraryOptions:

- Concurrent Library
- Concurrent Namebase
- Documents do not lock with parent
- Lock with parent

If no XDBLibraryOptions are specified explicitly, the default DDS Options will be used when a Location is created: Concurrent Library and Concurrent Namebase are set to true, and the other options are set to false.

The semantics of these options are described in the xDB documentation.

# Persistence Layer and File System

DDS includes a Filesystem implementation for the Persistence Layer, in the `com.emc.documentum.xml.dds.persistence.filesystem` package.

The following objects provide extra functionality on top of what is offered by the standard interfaces:

**FileSystemStore**

The FileSystemStore interface extends the Store interface, and adds accessors for :

- FileSystemType: the type of the File system
- Virtual Root: this allows one to define a Store which represents only a part of the FileSystem. By specifying a path, the directory with that path becomes the root of the Store. Any paths used when creating Locations or Containers in the Store will be treated as relative to the Virtual Root.
- Prefix: this represents the prefix that will be added to the absolute path, to make the path a valid path for the filesystem. For example, the prefix for a WindowsStore representing the

C-drive will be "C:". The prefix is generated automatically based on the Store configuration (i.e. the Id, Virtual Root and other properties).

**WindowsStore**

The WindowsStore models a Windows filesystem at the level of a disk - which means that a WindowsStore represents a disk, or a directory on the disk (if the Virtual Root is set). The Store Id should be the "disk letter" assigned in the OS, e.g. "C:" or "D:". The filesystem should be directly mounted on the Server running DDS.

**WindowsUNCStore**

The Windows UNC Store models a Windows share, mounted over the network. It has two specific properties :

- Host Name: specifies the hostname for the share
- Share Name: specifies the name of the share

**UnixStore**

The UnixStore models a Unix or Linux filesystem, or a directory on that filesystem (if the Virtual Root is set). The Store Id is null.

# Persistence Layer and XAM

DDS includes a XAM implementation in the `com.emc.documentum.xml.dds.persistence.xam` package.

**Note:** The XAM SDK and VIM Java libraries should be installed on the server running the DDS Server components, in order for the XAM integration to work.

The following objects provide extra functionality on top of what is offered by the standard interfaces:

**XAMStore**
The XAMStore object has some additional properties, with corresponding accessors:

- Connection String: the String used for connecting to a XAM device, specifying the VIM, hostname etc.
- Content Registry: the object which provides the virtual filesystem implementation. It keeps track of all existing Locations and Containers, since a XAM device has no hierarchical structuring of the data.

**CenteraStoreUser**

The CenteraStoreUser models a StoreUser for connecting to a XAM-enabled Centera. In addition to the id/password authentication model, it also supports PEA files, which can be used by Centera to authenticate a User.

The "PEA Filename" property specifies the filename of the PEA file to be used for the StoreUser. The file should be included on the classpath. To ensure this, the PEA files needed should be stored in the same directory as the Application Bootstrap file, and have the ".pea" suffix.

# Application Framework

Modeled as a Service to benefit from the Service lifecycle management facilities (see the Services Framework document), the Application framework provides for configuring, managing and accessing resources, such as Services, Stores, etc.

Administrators can start, pause, resume, stop and configure an Application without needing to take it offline.

The Application concept embodies the server component of an Application.

**Concepts**

The ApplicationProvider concept provides support for running an Application in a Java VM.

The ServiceManager concept represents the Application component for managing Services.

The StoreManager concept represents the Application component for managing Stores.

The StructureManager concept represents the Application component responsible for managing Structures, such as DataSets and Locales.

The XBaseManager concept represents the Application component responsible for managing XBases.

**Interfaces**

The Configuration concepts have been codified into interfaces in the `com.emc.documentum.xml.dds.application` package.

**Application Interface**
The Application interface (`com.emc.documentum.xml.dds.application.Application`) provides accessors to the following Application properties:

- Application Name represents the name of the Application, which is configurable.
- Main Store is the principal Store (see the Persistence Layer documentation) where the Application may keep its configuration, including Store configuration for other Stores, User information (see the User Service documentation) etc. On initialization, the Application connects to the Main Store to retrieve configuration information needed to carry out the initialization.
- Application User is the User (see the User Service documentation) which will be used by default for Application actions which need StoreUsers. The StoreUsers for this User should have administration rights in their respective Stores, unless they will not be used for administrative actions. Applications that use only one single StoreUser for all persistence operations can use the Application User for this purpose.
- Operation Manager is a reference to the OperationManager component which has been instantiated by the Application, and which can be used to execute Operations (see the Operation Framework documentation).
- Service Manager is a reference to the ServiceManager component which has been instantiated by the Application, and which can be used to access the Services configured for the Application (see the Service Framework documentation).
- Store Manager is a reference to the StoreManager component which has been instantiated by the Application, and which can be used to access the Stores configured for the Application (see the Persistence Layer documentation).
- XBase Manager is a reference to the XBaseManager component which has been instantiated by the Application, and which can be used to access the XBases configured for the Application (see the XBase Framework documentation).

The following methods are provided :

- getDefaultPersistor() and setDefaultPersistor() allow the developer to set and retrieve a default Persistor associated to a specified Store

- execute() is a convenience method, wrapping the corresponding method from the OperationManager
- getService() is a convenience method, wrapping the corresponding from the ServiceManager
- getStore() is a convenience method, wrapping the similar method from the StoreManager
- getXBase() is a convenience method, wrapping the similar method from the XBaseManager

**StoreManager Interface**

The StoreManager interface (`com.emc.documentum.xml.dds.application.StoreManager`) provides the following methods:

- getStore() retrieves the Store object with the specified alias
- getStores() returns a Collection containing all known Stores
- addStore() adds a Store to the StoreManager

**ServiceManager Interface**

The ServiceManager interface (`com.emc.documentum.xml.dds.application.ServiceManager`) provides the following methods:

- getService() for retrieval of Services corresponding to the supplied Service type or name
- createService() creates a new Service object of the specified type, using the specified class
- initializeServices() initializes all the Services, in a safe order, based on the dependencies that the Services declare. It returns a boolean indicating whether the Services were initialized successfully
- startServices() starts all the Services, in a safe order, based on the dependencies that the Services declare. It returns a boolean indicating whether the Services were started successfully
- pauseServices() pauses all the Services, in a safe order, based on the dependencies that the Services declare. It returns a boolean indicating whether the Services were paused successfully
- resumeServices() method will resume all the Services, in a safe order, based on the dependencies that the Services declare. It returns a boolean indicating whether the Services were resumed successfully
- stopServices() stops all the Services, in a safe order, based on the dependencies that the Services declare. It returns a boolean indicating whether the Services were stopped successfully

**StructureManager Interface**

The StructureManager interface (`com.emc.documentum.xml.dds.application.StructureManager`) has the following properties:

- Default DataSet: the default DataSet for the Application. No particular semantics apply to it. It is provided as a convenience for writing applications.
- Default Locale: for a DataSet, this is the default Locale for the specified DataSet for the Application. No particular semantics apply to it. It is provided as a convenience for writing applications. For example, the same Application could be made available with different URIs, with different Locales so that end users automatically get the data in the correct language.

The StructureManager interface provides the following methods:

- getDataSet() retrieves the DataSet with the specified alias.
- getDataSets() retrieves a list containing all the DataSets available to the Application.
- getDefaultDataSet() retrieves the default DataSet for the Application.
- addDataSet() makes the specified DataSet available to the Application through the specified alias. The DataSet must already exist and be available in the specified Store.
- createDataSet() methods create a new, empty DataSet.
- getLocale() methods retrieve the specified Locale from the specified DataSet.
- getLocales() methods retrieve a list of the available DDS Locales for the specified DataSet, optionally filtered on language or country.
- addLocale() makes an existing Locale available to the Application.
- createLocale() methods create a new Locale in the specified DataSet.

**XBaseManager Interface**
The XBaseManager interface (`com.emc.documentum.xml.dds.application.XBaseManager`) provides the following methods:

- getXBase() retrieves the XBase with the specified Id.
- getXBases() retrieves a List with all the XBases.
- addXBase() creates a new XBase with the specified Configuration.
- removeXBase() removes the XBase.

**Working with Applications**

Being a Service, an instantiated Application should be configured with the Bootstrap Configuration object (see the Configuration documentation), to obtain the information on the Main Store, as well as on the StoreUser to use with that Store. The Bootstrap can also contain references to the Services Configuration file, which contains configuration information for the Services used by the Application, and to the Stores Configuration file, which declares all the Stores used by the Application, along with their default StoreUsers.

When an Application is initialized by the Initialize action, the StoreManager, StructureManager, ServiceManager and XBaseManager are initialized, and Store, StoreUser, DataSet, Locale, Service and XBase objects are created, as specified in the Configuration.

When started, all Stores, Structures, Services and XBases are accessible through their respective Managers. The Application object can be obtained through the relevant ApplicationProvider methods, and can be used to retrieve the Stores, Structures, Services and XBases, either directly through the relevant convenience methods, or by retrieving the StoreManager, StructureManager, ServiceManager or XBaseManager and invoking their appropriate methods.

# Application Configuration

Application configuration uses the Serialization and Persistence functions to serialize and deserialize objects to and from XML. Configuration files are therefore serialized Java objects. There are two alternatives:

- If an object is simple, its serialized form can serve directly as either a configuration file, or part of a configuration file. For example: a Store object or StoreUser object can serve as its own configuration.
- For more complex objects, separate Configuration objects are available. The Application Bootstrap is an example of this.

**Concepts**

Generic concepts for configurability include:

- Configuration interface: signals that a Java object is used as a Configuration for some DDS entity. It offers no functionality.
- Configurable interface: is implemented by any Java object that can be configured with a Configuration.

The following concepts model Configurations for specific DDS entities:

- Bootstrap: the initial configuration needed by an Application. It contains:
  - the name of the Application
  - the configuration for the Main Store
  - references to the Configuration files for the Stores and the Services
- Stores Configuration: contains the Configuration for all the Stores accessible by the Application. It contains the Configurations for Store objects.

- Services Configuration: contains the Configuration for all the Services that are provided by the Application. It contains Service Configurations:
  - Each Service Configuration contains the configuration for a single Service.
- Structures Configuration: contains the configuration for all DataSets that are available to the Application.
- XBasesConfiguration contains the Configuration for the XBases defined for the Application.

**Interfaces**

The Configuration concepts have been codified into interfaces in the `com.emc.documentum.xml.dds.configuration` package.

**Configurable Interface**
The Configurable interface (`com.com.emc.documentum.xml.dds.configuration.Configurable`) provides accessors to the following Configurable object:

- configuration object: the object used for configuration of the Configurable

The Configurable interface provides the following methods:

- activateConfiguration(): applies the current Configuration to the object. The return value indicates whether the operation was successful.
- configure(): a convenience method, which sets the specified Configuration, and then activates it.

**Usage**

**Using the Application Bootstrap**

The example Application Bootstrap below is serialized with the DefaultDDSSerializer:

```
<bootstrap>
  <name>demo</name>
  <mainStore class="XDBStore">
    <alias>main</alias>
    <id>MyDatabase</id>
    <type>XDB</type>
    <defaultStoreUser
       class="XDBStoreUser">
      <id>Administrator</id>
      <password>secret</password>
      <administrator>true</administrator>
      <storeAlias>main</storeAlias>
    </defaultStoreUser>
    <bootstrap>xhive://localhost:1235</bootstrap>
    <cachePages>10000</cachePages>
  </mainStore>
  <serviceConfigurationReference>
    <path>/APPLICATIONS/garage/configuration</path>
    <name>Services.xml</name>
  </serviceConfigurationReference>
</bootstrap>
```

This Bootstrap example configures an Application with:

- name “demo”
- an XDB Store as Main Store, with “main” as Store Alias.

  The default StoreUser has “Administrator” as User Id, and “secret” as password.

  The URI of the XDB bootstrap file, which connects the application to the database, is “xhive://localhost:1235”.

The default StoreUser should be an administrator-level StoreUser in the Main Store, to allow it to perform Administrator functions, such as creating StoreUsers.

The example also references the Service Configuration file, with full path `/APPLICATIONS/garage/configuration/Services.xml`. The configuration files are always stored in the Main Store.

To actually configure the Application with this Bootstrap:
- deserialize this file into the Bootstrap object,
- instantiate an Application object,
- call the configure() method.

In the example below, the configXml variable is a String containing the bootstrap XML text:

```
Serializer<Object> serializer = new DefaultDDSSerializer();
Application application = new ApplicationImpl();

application.configure((Configuration<?>)serializer.deserialize(configXml));
application.fullStartup();
```

During initialization, the Application will resolve the Main Store, as well as the Services Configuration reference and the Stores Configuration reference, and initialize the ServiceManager and StoreManager with the resulting Configuration objects.

**Using the Application Bootstrap in Tomcat**

The DDS ApplicationStarter and SessionListener classes must receive the correct events when the Application is deployed or undeployed to the Apache Tomcat application server.

To ensure this, the following must be present in the applications's `web.xml` file before any servlet declarations:

```
<listener>
<listener-class>com.emc.documentum.xml.dds.servlet.ApplicationStarter</listener-class>
</listener>

<listener>
<listener-class>com.emc.documentum.xml.dds.servlet.SessionListener</listener-class>
</listener>
```

In the `web.xml` file, the *dds.application.config.path* context parameter should be set with the path of the bootstrap file as value. If not set, it defaults to “application-bootstrap.xml”, which will be read from the `WEB-INF` directory in the Tomcat directory that contains the deployed application.

The ApplicationStarter object will receive an event when the application is deployed in Tomcat, causing the object to read the Bootstrap configuration file, and to instantiate and start the Application.

Whenever a servlet session terminates, the SessionListener will perform the appropriate cleanup (e.g. logging out the user, if any).

**Using the Services Configuration**

When an Application starts, the Services Configuration is used to configure the ServiceManager class. Any Services that must be available should be configured in the Services Configuration.

The example below shows a basic Services Configuration, starting some standard Services:

```
<?xml version="1.0" encoding="UTF-16"?>
<services>
  <service>
    <type class="DDSServiceType">USER</type>
    <name>UserService</name>
<className>com.emc.documentum.xml.dds.user.impl.UserServiceImpl</className>

  </service>
  <service>
    <dependency>UserService</dependency>
    <type class="DDSServiceType">TOKEN</type>
    <name>TokenService</name>
<className>com.emc.documentum.xml.dds.user.impl.TokenServiceImpl</className>

  </service>
</services>
```

This example declares two Services:

- the User Service, named “UserService”, of ServiceType USER, with class “com.emc.documentum.xml.dds.user.impl.UserServiceImpl”
- the Token Service, named “TokenService”, of ServiceType TOKEN, with class “com.emc.documentum.xml.dds.user.impl.TokenServiceImpl”

Note that the TokenService declares a dependency on the User Service, which means that it will only be started after “UserService” has started, and that it will be stopped before “UserService” is stopped.

**Using the Stores Configuration**

When an Application starts, the Stores Configuration is used to start the StoreManager class. Any Stores that must be available should be configured in the Stores Configuration, except for the Main Store which has been defined in the Bootstrap file.

The example below shows a basic Stores Configuration, indicating two XDB Stores that will be available in the Application:

```
<stores>
  <XDBStore>
    <alias>first</alias>
    <id>MyDatabase</id>
    <type>XDB</type>
    <defaultStoreUser class="XDBStoreUser">
      <id>Administrator</id>
      <password>
        ...
      </password>
      <administrator>false</administrator>
      <storeAlias>main</storeAlias>
    </defaultStoreUser>
    <bootstrap>xhive://localhost:1235</bootstrap>
    <cachePages>10000</cachePages>
  </XDBStore>

  <XDBStore>
    <id>second</id>
    <name>YourDatabase</name>
    <type>XDB</type>
    <bootstrap>xhive://someother:1234</bootstrap>
    <cachePages>10000</cachePages>
```

```
  </XDBStore>
</stores>
```

The Store Alias is the alias that the Application will use to refer to the Store. It should be unique for the Application.

The Id, for an XDB Store, is the name of the database.

The Default Store User is the StoreUser used for accessing the Store if no specific StoreUser has been supplied. The other parameters are the same as in the Main Store declaration in the Bootstrap.

Other types of Store are supported as well :

- A WindowsStore is a Store mapping to a local Windows filesystem. An example configuration would be :

  ```
  <WindowsStore>
    <alias>testStore</alias>
    <id>C:</id>
    <type>FILESYSTEM</type>
    <virtualRoot>\my\location</virtualRoot>
    <fileSystemType>WINDOWS</fileSystemType>
  </WindowsStore>
  ```

  The Id field for a Windows Store should contain the drive letter. The virtual root is optional, and can be used to specify a directory that will act as the root directory of the Store. This means that if on this testStore, getLocation("\foo\bar") is invoked, the resulting Location would transparently be mapped to the C:\my\location\foo\bar directory.

- A UnixStore is a Store mapping to a local Unix filesystem. It has no Id. It can have a virtual root, like the WindowsStore.

  ```
  <UnixStore>
    <alias>the_alias</alias>
    <type>FILESYSTEM</type>
    <virtualRoot>/my/location</virtualRoot>
    <fileSystemType>UNIX</fileSystemType>
  </UnixStore>
  ```

- A XAMStore represents a XAM Storage Device. For now, the Centera is supported, through the CenteraStoreUser.

  ```
  <XAMStore>
    <alias>anAlias</alias>
    <id>anId</id>
    <type>XAM</type>
    <defaultStoreUser class="CenteraStoreUser">
      <administrator>false</administrator>
      <storeAlias>anAlias</storeAlias>
      <peaFileName>xamconnect.pea</peaFileName>
    </defaultStoreUser>
  <connectionString>snia-xam://centera_vim!127.0.0.1</connectionString>
  </XAMStore>
  ```

  The Connection String field should contain the connection string as defined by the XAM specification. The StoreUser should be a CenteraStoreUser. The peaFileName should be the name of the PEA file needed to connect to the Centera. The file itself should be stored in the same directory as the Application Bootstrap XML file.

**Using the Structures Configuration**

When an Application starts, the Structures Configuration is used to start the StructureManager class. Any DataSets that must be available should be configured in the Structures Configuration.

The example below shows a basic Structures Configuration, indicating two DataSets that will be available in the Application, the “garage” DataSet and the “britannica” DataSet:

```
<structures>
  <datasetref>
    <alias>garage</alias>
    <storeAlias>main</storeAlias>
    <id>garage</id>
    <defaultLocale>en_US</defaultLocale>
  </datasetref>

  <datasetref>
    <alias>britannica</alias>
    <storeAlias>main</storeAlias>
    <id>britannica</id>
    <defaultLocale>en</defaultLocale>
  </datasetref>

  <defaultDataSet>garage</defaultDataSet>
</structures>
```

The default DataSet for the application is the “garage” DataSet.

The “garage” DataSet has “en_US” as default Locale, and the “britannica” DataSet has “en” as the default Locale. The Default DataSet and Default Locale tags can be omitted, in which case no defaults will be defined for the Application.

**Note:**
Default Locale should not be specified for DataSets which are not locale-aware.

The Alias can be chosen freely, provided it is unique within the Application (i.e. no two configured DataSets should be configured with the same Alias).

The StoreAlias is the alias of the Store in which the DataSet resides. The Id is the Id of the DataSet in that Store. Within a Store, the Id must be unique. However, different Stores can contain DataSets with the same Id: the alias is used to distinguish them to the Application.

**Using the XBases Configuration**

When an Application starts, the XBases Configuration is used to start the XBaseManager class. Any XBases that must be available should be configured in the XBases Configuration.

The example below shows a basic XBases Configuration, indicating two XBases that will be available in the Application:

```
<xbases>
  <LogBase>
    <xBaseId>Rating</xBaseId>
    <storeAlias>main</storeAlias>
    <storeUserId>Administrator</storeUserId>
    <baseName>Rating</baseName>
    <location>/APPLICATIONS/myApp/XBase/</location>
    <strategy class="SingleFile"/>
  </LogBase>
</xbases>
```

This declares one LogBase, with Id "Rating". It is located in the store with StoreAlias "main".

The baseName is the prefix that will be used for the actual Documents created in the database - in this case, the name will be Rating.db.

The Location indicates where the LogBase files will be created.

The Strategy specifies the StorageStrategy : "SingleFile" means that all submitted entries will be stored in a single Document.

# Structures Framework

The Structures Framework provides concepts for the organization of data in the underlying Stores.

**Note:**

The Structure Framework API is experimental, and subject to changes in future releases. The most important feature for creating applications are the DDSDataSet and DDSLocale getLocation() and getContainer() methods, which will remain stable.

**Concepts**

The Structures Framework is built around the following major concepts:

A Structure models a set of data, together with a Structure Strategy, which determines where the data will be stored.

A Structure Strategy indicates which rules will be used for determining where in the Store data is persisted for a particular Structure.

A DataSet is a Structure which models a coherent collection of data. It can be locale-aware, in which case it can contain a number of Locales (see below).

A Locale is a Structure which models a subset of a particular DataSet, belonging to a single Locale. A Locale is either a language, a country, or a combination of these, with an optional variant for each case.

**Interfaces**

The Structures Framework concepts have been codified into interfaces in the `com.emc.documentum.xml.dds.structure` package.

**DDSStructure**

The DDSStructure interface (`com.emc.documentum.xml.dds.structure.DDSStructure`) provides accessors to the following DDSStructure properties:

- Structure Id: the Id of the Structure inside the Store where the Structure resides. It is unique to the Store, but not necessarily to the Application.
- Structure Strategy: the Strategy which determines where the data will be stored inside the Store.
- Root Location: the root location of the Structure.

The DDSStructure interface provides the following methods:

- getLocation(): creates a Location object inside the Structure, with the specified relative path.
- getContainer(): create a new Container object inside the Structure, with the specified relative path and Container Name.

**DDSDataSet**

The DDSDataSet interface (`com.emc.documentum.xml.dds.structure.DDSDataSet`) provides accessors to the following DDSDataSet properties:

- Alias: an alias unique to the Application, identifying the Structure. It is configured in the StructuresConfiguration.
- Locale Aware: indicates whether the DataSet can contain Locales.
- Default Locale: the default Locale for the DataSet. It is only relevant if the DataSet is locale-aware.

**DDSLocale**

The DDSLocale interface (`com.emc.documentum.xml.dds.structure.DDSLocale`) provides accessors to the following DDSLocale properties:

- Data Set: the DataSet to which the Locale belongs.
- Java Locale: the Java Locale (java.util.Locale) to which the Locale corresponds.

**Usage**

When working with DataSets and Locales, developers need not know the physical layout of the Store.

To access data, first obtain the relevant DataSet or Locale, using the application.getDataSet() and application.getLocale() methods.

These yield the relevant DDSDataSet and DDSLocale object, which have getLocation() and getContainer() methods for creating the correct Locations and Containers pointing to data in the DataSet or Locale, without having to know in which Store the DataSet is stored, or how that Store is laid out.

**Working with DDSDataSets**

To obtain a Location inside an DDSDataSet, the following code can be used:

```
DDSDataSet dataSet = Application.getDataSet("dataSetAlias");
Location location = dataSet.getLocation("foo/bar");
```

This yields a Location pointing to a physical path inside the Store.

- If the DataSet has been configured with the DDSStructureStrategy:

  `/DATA/dataSetId/foo/bar/`

- If the DataSet has been configured with the DocumentumStructureStrategy:

  `/DATA/dataSetId/Collection/foo/bar/`

To obtain a Container inside an DDSDataSet, the following code can be used:

```
DDSDataSet dataSet = Application.getDataSet("dataSetAlias");
Container container = dataSet.getContainer("foo/bar", "baz.xml");
```

This yields a Container pointing to a physical path inside the Store.

- If the DataSet has been configured with the DDSStructureStrategy:

  `/DATA/dataSetId/foo/bar/baz.xml`

- If the DataSet has been configured with the DocumentumStructureStrategy:

  `/DATA/dataSetId/Collection/foo/bar/baz.xml`

**Working with DDSLocales**

To obtain a Location inside a DDSLocale, the following code can be used:

```
DDSLocale locale = Application.getLocale("dataSetAlias", "en_US");
Location location = locale.getLocation("foo/bar");
```

This yields a Location pointing to a physical path inside the Store.

- If the DataSet has been configured with the DDSStructureStrategy:

  `/DATA/dataSetId/en_US/foo/bar/`

- If the DataSet has been configured with the DocumentumStructureStrategy:

  `/DATA/dataSetId/Collection/en_US/foo/bar/`

To obtain a Container inside an DDS Locale, the following code can be used:

```
DDSLocale locale = Application.getLocale("dataSetAlias", "en_US");
Container container = locale.getContainer("foo/bar", "baz.xml");
```

This yields a Container pointing to a physical path inside the Store.

- If the DataSet has been configured with the DDSStructureStrategy:

  `/DATA/dataSetId/en_US/foo/bar/baz.xml`

- If the DataSet has been configured with the DocumentumStructureStrategy:

  `/DATA/dataSetId/Collection/en_US/foo/bar/baz.xml`

# Store Locations

The root of a Store contains two Locations: DATA and APPLICATIONS.

- The APPLICATIONS Location contains a separate child Location, called an Application Root, for every Application that has this Store as its main Store. The name of the Location is the name of the application. Each Application Root contains all the resources and configuration for that Application.
- The DATA Location contains a separate child Location, called a DataSet Root, for every DataSet in the Store, as well as a Container for the configuration information for the DataSet, called a DataSet Configuration. The name of the Location is the name of the DataSet, and the name of the Container is the name of the DataSet, with ".xml" appended.

**Application root**

An Application Root contains the following Locations:

- A Configuration Location, called the Configuration Root, contains the configuration Containers for the Application, such as the Service configuration, Structure configuration, etc.
- A Users Location, called the Users Root, contains the User Home Locations, where user-specific data can be stored using the UserService. A User Home Location has the same name as the user. The Users Root also has one child Container for every user, whose name is the user name, with ".xml" appended, which contains configuration information about the user.
- A Resources Location, called the Resource Root, contains all the resources that are specific to the Application, and are used by the Application. This can include, but is not limited to, XProc pipelines, XSL stylesheets, XForms, etc. The internal organization of the Resource Root is up to the Developer.

**DataSet Root**

The organization of the contents of the DataSet Root depends on the StructureStrategy for the DataSet, and on whether the DataSet is locale-aware or not.

**Related links**

- Applications and Data

# DDS Locales and Java locales

A Java locale can be instantiated with a language and/or a country, and an optional variant. The DDS Locale conforms to this.

The DDS Locale name and Id correspond to the output of the toString() method of the corresponding `java.util.Locale` object. For example, a Locale for the US will have "_US" as name and Id. A Locale for US English will have "en_US" as name and Id.

See the javadoc documentation for `java.util.Locale` for more details.

# Storage examples

Suppose there is a data set called **DS1**, which includes a file `MyDocuments/Chapter1.doc`. The storage structure of the data set would be:

```
/DATA/DS1/Collection
/DATA/DS1/CollectionMetadata
```

**Without a locale**

If there is no locale, the file and its metadata will be stored under:

```
/DATA/DS1/Collection/MyDocuments/Chapter1.doc
/DATA/DS1/CollectionMetadata/MyDocuments/Chapter1.doc
```

**Note:** The metadata file gets the same file extension as the data file, even if it is an XML document.

**With a locale**

If there are two locales, en_US, and fr_FR, the file and its metadata will be stored under:

```
/DATA/DS1/Collection/en_US/MyDocuments/Chapter1.doc
/DATA/DS1/Collection/en_US/Metadata/MyDocuments/Chapter1.doc
```

and

```
/DATA/DS1/Collection/fr_FR/MyDocuments/Chapter1.doc
/DATA/DS1/Collection/fr_FR/Metadata/MyDocuments/Chapter1.doc
```

**With a locale and metadata**

If the folder `MyDocuments` included a metadata file, it would not be possible to place an XML document in the parallel structure. Its metadata file will be placed inside a nameless XML document:

```
/DATA/DS1/Collection/en_US/MyDocuments/Chapter1.doc
/DATA/DS1/Collection/en_US/Metadata/MyDocuments/Chapter1.doc
/DATA/DS1/CollectionMetadata/en_US/MyDocuments/{id:2}
```

and

```
/DATA/DS1/Collection/fr_FR/MyDocuments/Chapter1.doc
/DATA/DS1/Collection/fr_FR/Metadata/MyDocuments/Chapter1.doc
/DATA/DS1/CollectionMetadata/fr_FR/MyDocuments/{id:2}
```

where {id:2} represents the nameless XML Document.

In an XQuery, the connection between these files would be found through the use of the "XML Store builtin Metadata" and the "dds:subscription-element" key.

**Related links**

- XML Store Builtin Metadata

# Services Framework

The Service Framework provides a backbone structure for developing Services which can easily be managed at runtime. It provides a number of standard facilities for configuration and management.

**Concepts**

A Service is a facility which offers functionality within the context of an Application (see the Application Framework documentation). A Service can be used within an Application, as well as by other Services (for example, an E-mail Service could query the User Service to obtain the e-mail address of a User), and by clients.

A Service typically has a Java API codified in an interface. The Service APIs are extensible, and can be wrapped to expose Services to clients through GWT and other protocols (SOAP, RMI, WSDL, etc.). Service return objects are serializable.

Services have a built-in lifecycle model for managing Services at runtime, which even allows for temporarily taking a Service offline without needing to take the entire Application offline. The

lifecycle model provides for a succession of States and associated Actions. The model includes a number of "internal" states, which do not allow actions.

The following table summarizes the model, omitting “internal” States, for quick reference. For more information on States and Actions, see Services Lifecycle .

| State | Action(s) | Resulting state if action is succesful | Resulting state if action fails |
|---|---|---|---|
| Stopped | Initialize | Initialized | Stopped |
| Initialized | Start | Running | Initialized |
| Running | Pause<br>Stop | Paused<br>Stopped | Running<br>Running |
| Paused | Resume<br>Stop | Running<br>Stopped | Paused<br>Paused |

**Interfaces**

The Service Framework concepts have been codified into interfaces in the `com.emc.documentum.xml.dds.service` package.

**Service**

The Service interface (`com.emc.documentum.xml.dds.service.Service`) provides accessors to the following Service properties:

- Application: the Application object in the context of which the Service operates.
- Name: the name of the Service. This is configurable, and allows developers and/or administrators to provide the Service with a user-friendly name, for display in administration tools etc.
- Type: an enumerated constant which specifies the type of the Service.
- State: an enumerated constant specifying the State the Service is currently in.
- Last Action: the Action that was most recently performed on the Service, whether it was successful or unsuccessful.

The following methods are provided for control of the Service lifecycle:

- initialize(): performs the Initialize Action, and returns an indication of whether the Action was successful. It returns when the Service has entered either the Initialized or Stopped State.
- start(): performs the Start Action, and returns an indication of whether the Action was successful. It returns when the Service has entered either the Running or Initialized State.
- pause(): performs the Pause Action, and returns an indication of whether the Action was successful. It returns when the Service has entered either the Paused or Running State.
- resume(): performs the Resume Action, and returns an indication of whether the Action was successful. It returns when the Service has entered either the Running or Paused State.
- stop(): performs the Stop Action, and returns an indication of whether the Action was successful. It returns when the Service has entered either the Stopped, Running, Paused or Initialized State.
- fullStartup(): a convenience method to chain the Initialize Action (if successful) to the Start Action. It returns when the Actions have been performed successfully, or have failed. The resulting State can be Stopped, Initialized or Running.

**Implementation**

The abstract ServiceImpl class (`com.emc.documentum.xml.dds.service.impl.ServiceImpl`) provides for the basic implementation of a Service, and contains some additional abstract methods which should be implemented:

- checkDependencies(): should check whether all resources needed for initializing the Service are available. Typically at this time references to these resources are stored inside the Service if needed. The method returns a boolean indicating whether the check was successful. If not, the initialize() method, by which it is automatically called, will fail.
- executeInitialization(): should contain all processing which should take place in the Initializing State (except, of course, what is covered by the checkDependencies() method which is executed just before this method).
- executeStartup(): should contain all processing which should take place in the Starting State.
- executePause(): should contain all processing which should take place in the Pausing State.
- executeResume(): should contain all processing which should take place in the Resuming State.
- executeShutdown(): should contain all processing which should take place in the Stopping State.

**Implementing a Service**

To implement a Service, developers should proceed as follows:

1. Create the interface for the Service. The interface should contain all required public and administrative methods.
2. Create a subclass of ServiceImpl, which implements the interface from the previous step, implementing the abstract methods, providing the needed properties and adding the needed data structures.
3. Implement the interface methods.

**Using a Service**

Basic guidelines for using a Service:

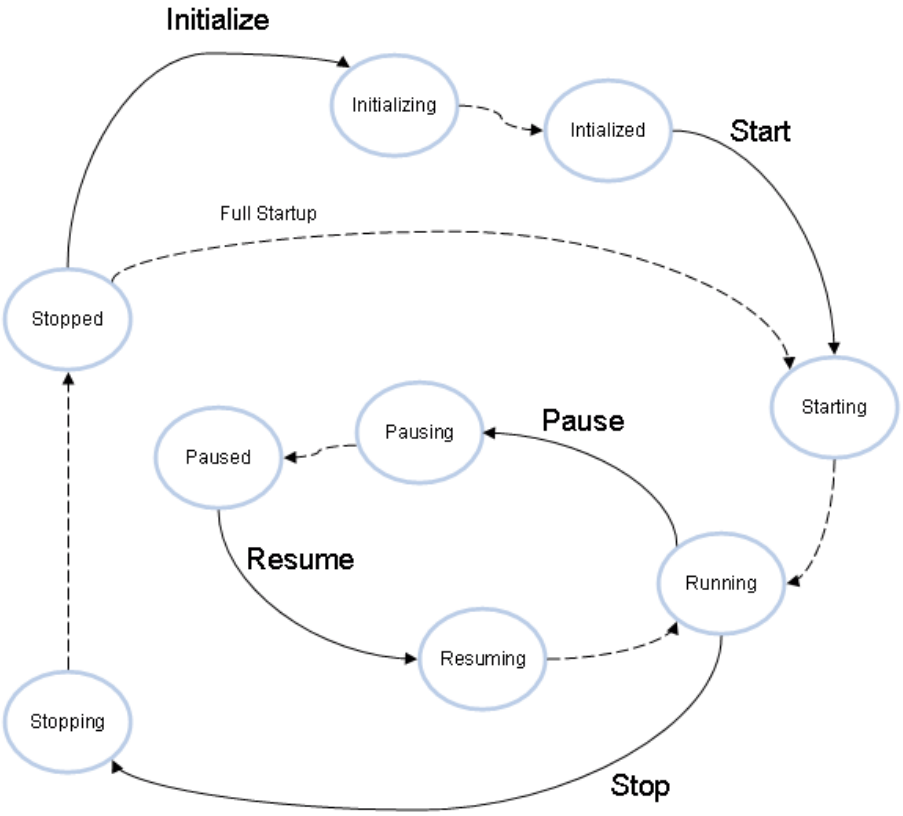
- To start a Service, it should first be configured. Then the initialize() and start() methods should be called.
- Always check whether called methods return true as expected.
- To configure a Service while it is active, it should be paused using the pause() method, after which configuration can be changed. When reconfigured, the Service can be resumed using the resume() method.
- To stop a Service, call the stop() method.
- To use one of the functions provided by a Service, simply call the appropriate method.

# Services Lifecycle

The lifecycle of a DDS Service is modeled by a state machine with the following actionable States: Stopped, Initialized, Running, Paused. In addition, a number of intermediate or "internal" states are provided, which do not allow actions.

Each Action allowed by an actionable State results in a State change if succesful.

The diagram below illustrates the relationships between the States and Actions in the lifecycle of a Service.




**Stopped**

Stopped State is the initial State of a Service when it has been created.

When Stopped, a Service can be configured.

Stopped State also results from a succesful Stop action, and from a failed Initialize action.

The only allowed action is the Initialize action, which changes the State to Initializing.

In the Stopped State, calling any of the “public” Service methods should normally result in a ServiceNotReadyException.

Calling configuration methods should succeed while the Service is Stopped. Administration methods may or may not be allowed, depending on whether or not they need initialized data structures. When not allowed, these methods should throw a ServiceNotReadyException.

**Initializing**

Initializing State results from the Initialize Action. In Initializing State, a Service will:

- check its dependencies (for example whether all resources needed by the Service are available)
- activate the configuration
- build the internal data structures

Allowed actions: none.

When the Service has successfully completed all tasks described above, its State automatically goes to Initialized. If any of these tasks fail (for example because a configuration file was not found, or a database was not accessible), the State automatically reverts to Stopped.

In the Initializing State, calling any of the Service methods should normally result in a ServiceNotReadyException.

**Initialized**

When Initialized, the Service is ready to be started. This State results from a succesful Initialize Action, and from a failed Start Action.

Allowed actions are Start and Stop, which change the State goes to Starting or Stopping respectively.

In the Initialized State, calling any of the Service methods should normally result in a ServiceNotReadyException.

**Starting**

When Starting, a Service will perform any processing necessary before going to the Running State. However, in many cases, no special processing takes place. Starting State is reached when the Start Action has been triggered.

Allowed actions: none.

When the Service has successfully completed processing, the State automatically goes to Running. If processing results in any failure, the State automatically reverts to Initialized.

In the Starting State, calling any of the Service methods should normally result in a ServiceNotReadyException.

**Running**

When Running, a Service is fully operational, and accepts and processes API method calls. In some cases, the Service may have its own thread, and perform continual processing, or accept requests which are performed asynchronously by the thread. Running State results from succesful Start or Resume Actions, and from failed Pause or Stop Actions.

Allowed actions are Pause and Stop, which will switch the State to Pausing and Stopping respectively.

**Pausing**

When Pausing, a Service will perform any processing necessary before going to the Paused State. However, in many cases, no special processing takes place. Paused State is reached when the Pause Action has been triggered.

Allowed actions: none.

When the Service has successfully completed processing, the State automatically goes to Paused. If processing results in any failure, the State automatically reverts to Running.

In Pausing State, calling any of the Service methods should normally result in a ServiceNotReadyException.

**Paused**

When Paused, a Service does not perform any processing, unless configuration or administrative Service methods are called. Paused state results from a succesful Pause Action, and from a failed Resume Action.

Paused State allows for reconfiguration or administration in a controlled way, without Stopping the Service, avoiding concurrency issues etc.

Allowed actions are Resume and Stop, which change the State to Resuming or Stopping respectively.

In the Starting State, calling any of the “public” Service methods should normally result in a ServiceNotReadyException. However, some configuration or administration methods may still be available.

**Resuming**

When Resuming, a Service will perform any processing necessary before going to the Running State. The Resuming State is reached when coming out of the Paused State.

Allowed actions: none.

When the Service has successfully completed processing, the State automatically goes to Running. If processing results in any failure, the State automatically reverts to Paused.

In Resuming State, calling any of the Service methods should normally result in a ServiceNotReadyException.

**Stopping**

When Stopping, a Service will perform any processing necessary before going to the Stopped State.

Allowed actions: none.

When the Service has successfully completed processing, the State automatically goes to Stopped. If processing results in any failure, the State automatically reverts to Running.

In Stopping State, calling any of the Service methods should normally result in a ServiceNotReadyException.

# Operation Framework

The Operation Framework facilitates the execution of server-side actions. It provides:

- Automatic Session management: Sessions do not need to be created, started or committed, the framework makes this transparent to the developer.
- Sequencing of Operations: the developer can bundle a number of Operations and have them executed in one Session.
- Support for transactions: transactions are supported, but only to the degree that the underlying Stores support them.
- Automatic deadlock retry looping: where a persistence action can fail due to a deadlock, a solution is to rollback and retry the transaction a number of times, until it succeeds. This is automated by the Operation Framework.

### Concepts

An Operation is an object which contains all the necessary information to execute an action on the server. For example, an Operation which is supposed to copy a document would specify the Store, Location and Container of both the source and target document.

An OperationExecutable is the executable counterpart to an Operation. It is a server-side object which ties the operation to an Application. It can access the information from the corresponding Operation, and execute the action.

The OperationManager is the Manager which provides the API for executing an Operation.

An OperationSequence is a sequence of Operations which will be executed as if it were a single Operation. It can contain a number of Operations (including other OperationSequences).

A SessionStoreUserStrategy is a Strategy which decides which StoreUser will be used for a particular User when executing an operation for that User which involves a Store. For example, all operations could be executed with the same StoreUser, or every User could have his own “personal” StoreUser. See the User Service documentation for more information about the User concept.

### Interfaces

The Operation Framework concepts have been codified into interfaces in the `com.emc.documentum.xml.dds.operation.framework` package.

### Operation

The Operation interface (`com.emc.documentum.xml.dds.operation.framework.Operation`) provides accessors to the following Operation properties:

- Operation Id: an identifier which can optionally be assigned to an Operation for subsequent identification. When executing an OperationSequence, this Id will be used to retrieve the Results from specific Operations within the OperationSequence, if needed.
- Read-Only: indicates whether the Operation is a read-only operation, if it involves any Stores. This will allow the framework to optimize any involved transactions.
- Executable Class Name: the class name of the OperationExecutable which corresponds to this Operation, and which will be instantiated by the Operation Framework when the Operation is submitted for execution. If the value of this property is null, the classname of the Operation is used, with the String “Executable” appended. For example, if the Operation has classname com.foo.BarOperation, the framework will assume the classname of the corresponding OperationExecutable to be com.foo.BarOperationExecutable.
- Store Ids: the list of Ids of all the Stores which are involved in the Operation.

The following methods are provided for control of the Service lifecycle:

- initialize(): performs the Initialize Action, and returns an indication of whether the Action was successful. It returns when the Service has entered either the Initialized or Stopped State.
- start(): performs the Start Action, and returns an indication of whether the Action was successful. It returns when the Service has entered either the Running or Initialized State.
- pause(): performs the Pause Action, and returns an indication of whether the Action was successful. It returns when the Service has entered either the Paused or Running State.
- resume(): performs the Resume Action, and returns an indication of whether the Action was successful. It returns when the Service has entered either the Running or Paused State.
- stop(): performs the Stop Action, and returns an indication of whether the Action was successful. It returns when the Service has entered either the Stopped, Running, Paused or Initialized State.
- fullStartup(): a convenience method to chain the Initialize Action (if successful) to the Start Action. It returns when the Actions have been performed successfully, or have failed. The resulting State can be Stopped, Initialized or Running.

**OperationExecutable**

The OperationExecutable interface (`com.emc.documentum.xml.dds.operation.framework.OperationExecutable`) provides accessors to the following OperationExecutable properties:

- Application: a reference to the Application object in the context of which the Operation is executed.
- Operation: a reference to the Operation object for whose execution the OperationExecutable was created.
- Can RollBack: indicates whether the OperationExecutable can be rolled back. This is only true if the OperationExecutable contains specific rollback code, and should never be true if the Store has proper transaction support, since the rollback will then take place at Session level.

The OperationExecutable interface provides the following methods:

- beforeRun(): to be executed before the action proper is executed. It is provided to create superclasses with common setup behaviour needed before the action proper is executed, and which may need to be extended or overridden.
- afterRun(): to be executed after the action proper is executed. It is provided to create superclasses with common teardown behaviour needed after the action proper is executed, and which may need to be extended or overridden.
- run():executes the action, and returns the Result, or throws an OperationFailedException if the Operation could not be executed.
- rollback(): will roll back the Operation, if it supports rollback.

**OperationManager**

The OperationManager interface (`com.emc.documentum.xml.dds.operation.framework.OperationManager`) provides the following method:

- execute(): executes an Operation and returns the Result..

**SessionStoreUserStrategy**

The SessionStoreUserStrategy interface (`com.emc.documentum.xml.dds.operation.framework.SessionStoreUserStrategy`) provides accessors to the following OperationExecutable properties:

- Fallback Strategy: a contained SessionUserStrategy which can be configured as fallback. If the Strategy cannot determine a StoreUser, the fallback Strategy will be used instead.

The SessionStoreUserStrategy interface provides the following method:

- getStoreUser(): returns the StoreUser which should be used for a given Application, User and Store. If no such StoreUser can be determined by the Strategy, the result from querying the fallback Strategy will be returned.

**Implementation**

The `com.emc.documentum.xml.dds.operation.library` package contains several sub-packages which provide implementations for its concepts:

- `com.emc.documentum.xml.dds.operation.library.basic`
- `com.emc.documentum.xml.dds.operation.library.persistence`
- `com.emc.documentum.xml.dds.operation.library.result`

The `com.emc.documentum.xml.dds.operation.library.basic` package contains a number of abstract classes which partially implement some useful Operation types:

- The AbstractOperation class provides a partial implementation which implements the accessors for the Operation Id property.

- The AbstractNoStoreOperation, AbstractSingleStoreOperation, AbstractTwoStoreOperation and AbstractMultiStoreOperation extend AbstractOperation, and provide accessors and constructors for Operations which need no, one, two or many Stores respectively.

It also contains the OperationSequence and OperationSequenceExecutable objects:

- The OperationSequence is a complete Operation implementation, and provides accessors for the list of contained Operations.

The `com.emc.documentum.xml.dds.operation.library.persistence` package contains the Operations corresponding to the actions in the Persistence Layer. This package contains one Operation for every action offered by the Persistor in the Persistence Framework.

The necessary parameters are provided in the different constructors. For every constructor, an alternative constructor is provided which takes a Persistor as argument, to enable the use of custom Persistors.

The `com.emc.documentum.xml.dds.operation.library.result` package contains the various Result implementations. This package contains the various Results that Operations can return, including:

BlackBoardResult is returned by an OperationSequence. The value() method returns a Map whose keys are the Ids of the contained Operations, and whose values are the Results of those Operations. If the Id of an Operation was null, its result is not included in the BlackBoardResult.

**Usage**

**Working with Operations**

All that is needed, is to call the execute() method on the OperationManager, providing a User as well as the actual Operation. The following example creates a Location in a Store with an Operation:

```
String storeId = …;
Location location = …;
User user = …;
OperationManager.execute(user, new CreateLocationOperation(storeId,
location, false));
```

When executing the Operation, the Framework will:

- Instantiate the OperationExecutable corresponding to the Operation
- Create a Session for each Store involved, for the User
- Start the Sessions
- Execute the beforeRun() method of the OperationExecutable
- Execute the run() method of the OperationExecutable, supplying the Sessions
- Execute the afterRun() method of the OperationExecutable
- Commit the Session
- Return the Result

**Working with OperationSequences**

The following example shows how to work with an OperationSequence:

```
OperationSequence sequence = new OperationSequence();
sequence.setId("SeqId");
sequence.addOperation("CreateLocation1", new
CreateLocationOperation(this.storeId, location1, true));
sequence.addOperation("CreateLocation2", new
CreateLocationOperation(this.storeId, location2, true));
BlackBoardResult result = (BlackBoardResult)OperationManager.execute(user1,
 sequence);
```

Basically, the Operation Framework behaves much like a single Operation, with some additions:

- If an Operation fails, all the Operations will be rolled back, but in reverse order.

- If an Operation fails, all Sessions will be rolled back.
- An OperationSequence always returns a BlackBoardResult.

**Note:**

If all of the Operations use one and the same Store, and that Store supports transactions, the entire OperationSequence will take place in a single transaction, and rollback will amount to rolling back that transaction, and should never cause any issues.

However, if multiple Stores are used and an Operation fails, the rollback will be a “best effort”: due to the lack of a two-phased commit, the Stores may be in an undefined state – one Sessions may have been committed, and another one may have been rolled back.

# User Service

The User Service facilitates user management for Applications. It includes facilities for:

- Creating and deleting Users.
- Retrieving User information.
- User login and logout.
- Password authentication.
- Automatic deadlock retry looping: where a persistence action can fail due to a deadlock, a solution is to rollback and retry the transaction a number of times, until it succeeds. This is automated by the Operation Framework .

The User Service will be started automatically on Application startup, provided it has been configured for the Application (see Application Configuration for more information).

**Concepts**

A User models the end user of an Application (see Application Framework for more information). It is used for authentication and authorization at the DDS level.

A User is only valid for a single Application. If end user access is needed for multiple Applications, a separate User should be created for every Application.

If a single User of an Application needs access to several Stores, the User needs to be mapped to several StoreUsers (see Persistence Layer for more information).

The information for a User is stored in the Main Store of the Application. Each User has a Home Location where any objects created by the User will be persisted. This Home Location is also stored in the Main Store at:

`/APPLICATIONS/<applicationName>/users/<userId>` .

The User information (id, password, StoreUsers, etc) is stored in the file `/APPLICATIONS/<applicationName>/users/<userId>.xml`.

**Note:** StoreUsers are not managed by the User Service. The Persistence Operations in the Operation Framework provide the necessary functionality for creating StoreUsers.

A User Token is a token which is obtained when a User logs into an Application. It uniquely identifies the “DDS Session” which is the period between logging in and logging out (either because the User logs out actively, or because a timeout occurs).

The Token Service manages User Tokens.

**Interfaces**

The User Service concepts have been codified into interfaces in the `com.emc.documentum.xml.dds.user` package.

**User**

The User interface (`com.emc.documentum.xml.dds.user.User`) provides accessors to the following User properties:

- User Id: the identifier assigned to the User, and used for uniquely identifying the User in the context of an Application.
- Password: the password the User will need for log in.
- Administrator: indicates whether the User is an administrator for the Application.
- Default StoreUser: the StoreUser which can be used to access a particular Store. It is possible to set and retrieve a Default StoreUser for each Store.

The following method is provided:

- checkPassword(): returns true if the provided password is the correct password for the User.

**UserService**

The UserService interface (`com.emc.documentum.xml.dds.user.UserService`) provides accessors to the following User property:

- Users Location: the Location where all the User information is stored for the Application. This defaults to the Location with path `/APPLICATIONS/<applicationName>/users` in the Main Store.

The UserService interface provides the following methods:

- getUsers(): for retrieval of Users as a Collection.
- getUser(): for retrieval of a User by User Id.
- getHome(): for retrieval of the Home Location of a User based on the User Id.
- createUser(): for creation of Users.
- deleteUser(): for deletion of Users.
- loginUser(): for logging in of a User.
- logoutUser(): for logging out of a User.
- isLoggedIn(): indicates whether the User with the specified User Id is currently logged in.

**UserToken**

The UserToken interface (`com.emc.documentum.xml.dds.user.UserToken`) provides accessors to the following UserToken properties:

- Application Name: the name of the Application to which the User logged in.
- User Id: the User Id of the User who logged in.
- Token Id: a unique identifier distinguishing UserTokens for which the other properties happen to be the same.
- Timestamp: the java timestamp at which the User logged in.

**TokenService**

The TokenService interface (`com.emc.documentum.xml.dds.user.TokenService`) provides the following methods:

- createToken(): creates a new UserToken based on the specified User and Application.
- getApplication(): retrieves the Application object in whose context the Token was created.
- getUser(): retrieves the User object for which the Token was created.

**Usage**

To use the UserService, a reference is obtained from the Application object:

```
UserService userService =
(UserService)application.getServiceManager().getService(DDSServiceType.USER);
```

Thereafter, usage consists of invoking the methods on the Service.

# Response Service

The Response Service is used to deal with information provided by Users, such as ratings, comments, ... The Response Service deals with the reception and processing of information supplied by the User.

**Concepts**

A Response is an XML fragment which contains information the User entered, and which is submitted to the server. The Response could be a rating of a document, a comment, a blog post, etc.

A PreProcessor is an object which will process, and potentially transform, the Response before it is stored in an XBase.

A PostProcessor is an object which will process the Response after it has been stored in an XBase.

A ResponseFilter is an object which can be associated with processors (through ProcessorAssociations) and XBases (through XBaseAssociations), determining whether the processor should be applied to the Response, or whether the Response should be stored in the XBase.

When a Response is submitted, the following steps are taken:

- The Service iterates over the PreProcessors in the order in which they were registered, checking whether the Response should be processed with the associated ResponseFilter. If so, the Response is processed and potentially transformed. If transformation occurs, it is the transformed Response that will be used from then on.
- The Service looks up the first registered XBase for which the ResponseFilter accepts the Response. The Response is stored in that XBase. If there is no matching Filter, the Response will not be stored.
- The Service iterates over the PostProcessors in the order they were registered, checking whether the Response should be processed with the associated ResponseFilter. If so, the Response is processed and potentially transformed. If transformation occurs, t is the transformed Response that will be used from that point onwards.

**Interfaces**

The Response Service concepts have been codified into interfaces in the `com.emc.documentum.xml.dds.usi` package.

**User**

The User interface (`com.emc.documentum.xml.dds.usi.Response`) provides accessors to the Response content:

- The asNode() and asString() methods return the content either as a DOM node or as a String representing the XML fragment.
- The Request Id is a String identifying the Request to which this is a Response.
- The Context Id is a String identifying the context in which the Response was given. Typically, if the Response concerns a document or item, the Context Id would be the Id of that document or item.
- The Time field contains the time at which the Response was submitted.

The SimpleResponse class is a simple implementation of this interface, which can be instantiated using either an XML Fragment or a DOM Node.

**ResponseService**
The ResponseService interface (`com.emc.documentum.xml.dds.usi.ResponseService`) provides the following methods:

- submit() methods: allow submission of a Response, as a String, DOM Node or using an existing Response object. Submission will trigger preprocessing, storage and postprocessing.
- registerPreProcessor() and registerPostProcessor(): these methods set up an association between the supplied Filter and the supplied Processor. If a submitted Response matches the Filter, the corresponding Processor will be executed.
- registerXBase(): sets up an association between the supplied Filter and the supplied XBase. If a submitted Response matches the Filter, it will be stored in the corresponding XBase.
- registerXBase() methods: allow the deactivation of the specified processors or XBases.

**ResponseFilter**
The ResponseFilter interface (`com.emc.documentum.xml.dds.usi.ResponseFilter`) provides the following method:

- accept(): returns true if the tested Response is accepted by the Filter, and false if it is not.

**ResponseProcessor**
The ResponseProcessor interface (`com.emc.documentum.xml.dds.usi.ResponseProcessor`) provides the following method:

- process(): processes the Response, and returns the Response, or the transformed Response if the processor implements a transformation.

**Configuring the ResponseService**

To configure the ResponseService, the `Services.xml` file must be edited. The following example illustrates the XML fragment that should be inserted :

```
<responseservice>
    <type class="DDSServiceType">RESPONSE</type>
    <name>ResponseService</name>

<className>com.emc.documentum.xml.dds.usi.internal.ResponseServiceImpl</className>

    <preProcessors/>
    <postProcessors>
      <processorAssociation>
        <filter
class="com.emc.documentum.xml.dds.demo.server.AllPassFilter"/>
        <processor
class="com.emc.documentum.xml.dds.demo.server.AverageCalculator"/>
      </processorAssociation>
    </postProcessors>
    <xBases>
      <xBaseAssociation>
        <filter
class="com.emc.documentum.xml.dds.demo.server.AllPassFilter"/>
        <xBaseId>Rating</xBaseId>
      </xBaseAssociation>
    </xBases>
  </responseservice>
```

In addition to the normal Service configuration parameters (type, name and className), 3 new sections are inserted :

- preProcessors contains a list of ProcessorAssociations, associating a filter with a processor.
- postProcessors contains a list of ProcessorAssociations, associating a filter with a processor.
- xBases contains a list of XBaseAssociations, associating a filter class with an xBase Id.

The class attributes should contain the full class name of the processor or filter class. If the processor or filter has any additional configuration, they will be present as child elements. What these are depends on the processor or filter.

The XBases should have been defined in the XBases configuration file.

**Usage**

**Response Content**

The XML in the Response can have any structure, but for typical Responses we strongly recommend the following simple structure :

```
<entry>
  <requestId>[Request Id]</requestId>
  <contextId>[Context Id]</contextId>
  <time>[Submission Time]</time>
  <key1>[Value 1]</key1>
  <key2>[Value 2]</key2>
  ...
</entry>
```

If possible, the Response should be modeled as a set of key-value pairs, where the keys are used as the tagnames of the child elements of the root entry element, and the values are stored in the content of those child elements.

By sticking to this simple structure, it will be possible in future versions to take advantage of some additional functionality in for instance the LogBases into which Responses can be stored. These functions will expect this type of XML structuring.

The recommended fields are:

- requestId: a String identifying the original Request which triggered the Response. This Request is the logical representation of the question the User is answering (possibly implicitly).

  For example, if many objects on a site are being rated, the String “rating” could be used as requestId to show that the Response is actually a rating applied by the User to something on the site.

- contextId: a String identifying the context in which the Request was answered.

  For example, if a book is being rated, the context Id could be the ISBN or other identifier for the book. Similarly, if a webpage is being rated, the URL could be used as the context identifier.

- time: the time at which the response was submitted. This should be formatted using the xsd:dateTime format from XML Schema.

**Storing Responses**

The primary purpose of the Response Service is to store information received from Users. To do this, XBases should be created or configured (see the XBase chapter), and registered to the ResponseService with the appropriate filters.

For example, if all the Responses with Request Id “foo” should be stored in XBase “bar” (which has been preconfigured for the Application in the XBaseManager), a Filter could be created with the following accept() method :

```
public boolean accept(Response response) {
  return "foo".equals(response.getRequestId());
}
```

The ResponseService will store all Responses with Request Id “foo” in the “bar” xBase through invoking:

```
XBase xBase = application.getXBase("bar");
responseService.registerXBase(myResponseFilter, xBase);
```

**Preprocessing Responses**

Use preprocessing if there is a need for normalization, or some other transformation, before a Response can be stored.

Other operations can be performed during preprocessing, but if no transformation is intended, they should probably be executed as postprocessing. This allows the Response to be stored as quickly as possible.

An implementation of a ResponseProcessor could look something like:

```
Response process(Response response) {
  String oldXml = response.asString();
  String newXml = [... transform the XML ...];
  return new SimpleResponse(newXml);
}
```

A PreProcessor is registered by executing :

```
responseService.registerPreProcessor(myResponseFilter, myProcessor);
```

**Note:** Keep in mind that, when a Response has been processed by a PreProcessor or a PostProcessor for further processing (i.e. filter checking, processing and storage in the Response Service), it is replaced by the Response received from the process() call.

# Logging Framework

The Logging Framework provides some facilities for server-side logging.

**Concepts**
The Logging Framework is built around the following concepts:

- The LogCenter object provides all the configuration options, as well as the methods for logging messages and exceptions.
- A Logger outputs logged messages to a configured output medium, such as standard out, or a file. Separate Loggers can be registered to catch the output from different Java classes.

Initially, without any configuration, the LogCenter will log everything to a SystemStreamsLogger, which logs all the messages to STDOUT.

A Default Logger can be registered, which will log everything to a file. Several classes can be redirected to a single Logger. Additional Loggers can be registered for specified class names. Whenever log messages have such a class specified as the sender, the messages will be redirected to the appropriate Logger.

For these loggers, it is possible to override the global **Debug Status** (which indicates whether debug-level messages will be logged). In practice, this means that it is possible to redirect logging for subsystems or particular classes to separate log files. This can be done at runtime.

**Interfaces**

The LogCenter has been codified into interfaces in the `com.emc.documentum.xml.dds.logging` package.

**Operation**

The LogCenter interface (`com.emc.documentum.xml.dds.logging.LogCenter`) provides accessors to the following LogCenter properties:

- Default Logger: the Logger to which all messages will be routed if no sender is specified for the message.

- Default Log Path: the path where log files will be created, if a FileLogger is used.
- Default Log Suffix: the suffix for log filenames, when a FileLogger is used.
- Debug Status: indicates whether messages at DEBUG level will be logged for classes for which no explicit Logger is registered.

The LogCenter interface provides accessors to the following methods:

- log(): log a message at LOG level.
- warning() methods: log a message at WARNING level.
- error() methods: log a message at ERROR level.
- exception() methods: log an exception and an optional message at EXCEPTION level.
- debug() methods: log a message at DEBUG level.
- activateDebug() and deactivateDebug() methods: set the Debug Status to true or false respectively.
- register(String className, String prefix, boolean debugLevel): adds a new FileLogger for the specified class. Prefix will be used as to construct the filename (see the Usage section below). Debug Level specifies whether DEBUG-level messages from this class should be logged.
- setDebug(): allows activation or deactivation of debug mode for specific classes.
- unregister(): removes the custom mapping for the specified class. Logging for that class will be directed to the Default Logger.
- shutdown(): closes all the Loggers properly. LogCenter should not be used after shutdown() has been called.

**Usage**

**Working with Logcenter**

To log a message, from any class, just call:

```
LogCenter.log(this, "A message from me");
LogCenter.warning(this, "A warning from me");
LogCenter.error(this, "An error from me");
```

**Log Levels**

Logging supports 5 “levels” of logging : LOG, WARNING, ERROR, EXCEPTION and DEBUG.

A message should be logged at LOG level for routine events that should be logged every time they occur. For example, for auditing purposes, when a user logs in the time and user id could be logged.

The WARNING, ERROR and EXCEPTION levels can be used to signal the occurrence of unusual events which may require investigation by an Administrator.

The DEBUG level can be used for in-depth logging about what the application code is doing, and can be useful during development, as well as for troubleshooting in a production environment.

**Debug Mode**

Messages at DEBUG level are logged on the following conditions:

- If the Debug Status of the LogCenter is true, the message will always be logged.
- If the Debug Status is false, but a sender has been specified, and the Debug Status for the class of the sender has been set to true, the message will be logged.

In all other cases, the message will not be logged.

**Log Files**

If no configuration is performed, the LogCenter will send all messages to STDOUT by default.

To redirect all logging to a file, a new FileLogger should be created using the FileLogger(String logPath, String logPrefix, String logSuffix) constructor, and this should be set as the DefaultLogger.

The following example will create a log file with full path :

```
/<logPath>/<logPrefix>_<timestamp>.<logSuffix>
```

The following example:

```
LogCenter.setDefaultLogger(new FileLogger("/temp/log/", "DDS_", ".log"))
```

if executed on July 3rd, 2008 at 8h23, will redirect all logging to the file:

`/temp/log/DDS_20080703_082356.log`

To redirect all logging from classes `com.my.Foo and com.my.Bar` to a separate file called Fubar, you can register a new Logger, after ensuring that the default suffix and default path have been set correctly on the LogCenter :

```
LogCenter.setDefaultLogPath("/temp/log");
LogCenter.setDefaultSuffix(".log");
LogCenter.register("com.my.Foo", "Fubar", true);
LogCenter.register("com.my.Bar", "Fubar", true);
```

If executed on July 3rd, 2008 at 9h00, this will redirect all logging to the file:

`/temp/log/Fubar_20080703_090000.log`

All these operations can be performed on a running system. The new configuration will take effect immediately.

# XBase Framework

The XBase framework provides a simple and easy way to store data in xDB Stores.

**Concepts**

An XBase is a logical data structure, in which one can store XML Fragments submitted as XBaseEntries.

Underlying the XBase is a set of XML documents, called XBaseFiles. These files store the actual data, using a StorageStrategy.

The StorageStrategy can be implemented in many ways, for example:

- It can maintain a number of documents, using round-robin to write XBaseEntries into them, improving performance through multi-threaded access.
- It can use some identifier in the XBaseEntry to determine in which file the Entry should be stored – thus grouping similar Entries in a single document.

The LogBase is an implementation that covers the "append-only" case, which means it behaves much like a logfile (or a set of log files) to which data are appended, but no updates are performed.

To manage the XBases, the XBaseManager provides for configuration and retrieval of XBases. The XBaseManager and its configuration are described in the Application Framework documentation.

**Interfaces**

The Structures Framework concepts have been codified into interfaces in the `com.emc.documentum.xml.dds.xbase` package.

**XBase**

The XBase interface (`com.emc.documentum.xml.dds.xbase.XBase`) provides accessors to the following XBaseproperties:

- Id: the unique identifier of the XBase.
- Store: the XML Store in which the XBase resides.

The XBase interface provides the following methods:

- store(): stores the specified XBaseEntry in the XBase.
- clear(): deletes all Entries that have been stored in the XBase.
- newEntry(): act as factory methods for creating the appropriate type of XBaseEntry, based on a String representing the XML Fragment, or a DOM Node.
- newXBaseFile(): acts as a factory method for creating new XBaseFiles. Should only be used by the StorageStrategy, when creating a new document.

**XBaseEntry**

The XBaseEntry interface (`com.emc.documentum.xml.dds.xbase.XBaseEntry`) provides the following method:

- write(): tells the XBaseEntry to write itself into the provided XBaseFile.

**XBaseFile**

The XBaseFile interface (`com.emc.documentum.xml.dds.xbase.XBaseFile`) provides the following methods:

- getEntryCount(): returns the number of Entries stored in the XBaseFile.
- create(): creates the actual underlying XML document in the database.
- store(): stores the specified XBaseEntry in the XBaseFile.
- clear(): deletes all the Entries stored in the XBaseFile.
- close(): closes the XBaseFile, after which no more Entries should be stored in this file.

**LogBase**

The LogBase interface (`com.emc.documentum.xml.dds.xbase.logbase.LogBase`) adds the following method to the XBase interface:

- newEntry(): provides a way to create a LogBaseEntry based on a Map containing key-value pairs.

**LogBaseFile**

The LogBaseFile interface (`com.emc.documentum.xml.dds.xbase.logbase.LogBaseFile`) adds the following methods to the XBaseFile interface:

- addAsString() and addAsNode(): these methods take an XML Fragment, coded as a String or a DOM Node, and add it to the LogBaseFile document, by appending it as a child of the root element.
- additional methods for use by the LogBaseEntry write() method, and allow the Entry to write itself into the LogBase. For a detailed description, please see the Javadoc for LogBaseEntry (com.emc.documentum.dds.xbase.logbase.LogBaseEntry).

**Usage**

The following example shows how to submit an Entry as an XML Fragment:

```
XBaseEntry entry =
xBase.newEntry("<entry><name>foo</name><rating>5</rating></entry>");
xBase.submit(session, entry);
```

**Note:** It may be easier to use the StoreEntryOperation in the Operations Framework, since that takes care of managing the needed Sessions for the Store.

It is recommended to keep the XML content format simple. For LogBases, the recommended format is:

```
<entry>
 <key1>value1</key1>
 <key2>value2</key2>
 <key3>value3</key3>
 ...
</entry>
```

This recommended format has a root element called “entry” and encodes the data as key-value pairs, with the key used as tagname for a child element, and the value being the content of that element. Future extensions to LogBases may assume this format is being used.

# XProc Service

The XProc Service provides the functionality of the XProc processor to DDS Applications.

The XProc Service will be started automatically on Application startup, provided it has been configured for the Application (see Application Configuration and XProc Service Configuration for more information.).

**Concepts**

XProc represents a wrapper object on top of the XProc processor that provides for integration with DDS and its frameworks.

Resource manipulation in the XProc implementation is URI-based.

The XProc processor uses so-called resolver modules for resolving content for read access. When resolving a URI, the processor consults all resolver modules until it finds one that understands the specified URI scheme. If no resolver module supporting the given URI scheme is found, a resolution error occurs.

For storing content, the XProc processor relies on so-called writer modules. Writer modules are similar to the resolver modules in the sense that they are used for providing access to target locations represented by a URI. When resolving a target URI, the processor consults all writer modules until it finds one that understands the specified URI scheme. If no writer module supporting the given URI scheme is found, a writer error occurs.

Any number of custom resolver and writer modules can be registered with the XProc implementation. By default, XProc contains reader module and writer module implementations that support DDS URIs .

An XProc instance is always bound to a particular DDS User . All resource manipulations (both read and write) are performed using this user information.

When referring to resources, the XProc Service needs to be able to create Sessions for the Persistence Layer framework. To facilitate that, each XProc instance is assigned a session pool which it can use for creating new sessions and for reusing existing sessions.

**Interfaces**

The XProc Service concepts have been codified into interfaces in the `com.emc.documentum.xml.dds.xproc` package.

**DDSXProc**

The XProc interface (`com.emc.documentum.xml.dds.xproc.DDSXProc`) provides accessors to the following:

- XProc processor: the XProc processor itself. The processor provides APIs for creating a new XProc pipeline, executing a pipeline, etc.
- Session pool: the session pool provides the XProc resolver and writer modules with the ability to create and reuse Sessions. The session pool also keeps track of all Sessions that have been created while interacting with the XProc processor.
- Open sessions: the collection of all Sessions that have been created while interacting with the XProc processor.

**XProcService**

The XProcService interface (`com.emc.documentum.xml.dds.xproc.XProcService`) provides the following method:

- newXProc(): this method returns an XProc object that represents a DDS-aware wrapper on top of the XProc processor itself.

**AbstractDDSResolverModule**
Abstract implementation of the XProc resolver interface (`com.emc.documentum.xml.dds.xproc.AbstractDDSResolverModule`) that provides basic integration with DDS. Extend this class if you want to implement a custom resolver module. The DDS-aware resolver module provides accessors to the following:

- DDS application: the Application object.
- DDS user: the User using the XProc processor.
- Session pool: the session pool associated with the XProc processor.

**AbstractDDSWriterModule**
Abstract implementation of the XProc writer interface (`com.emc.documentum.xml.dds.xproc.AbstractDDSWriterModule`) that provides basic integration with DDS. Extend this class if you want to implement a custom writer module. The DDS-aware writer module provides accessors to the following:

- DDS application: the Application object.
- DDS user: the User using the XProc processor.
- Session pool: the session pool associated with the XProc processor.

**XProcUtils**

This class contains miscellaneous XProc-related utility methods.

**Usage**

To use the XProcService, a reference is obtained from the Application object:

```
// Get the XProc service instance
XProcService xprocService =
(XProcService)application.getServiceManager().getService(DDSServiceType.XPROC);

User user = ...;
boolean readOnly = true;

// Create a new DDSXProc instance
DDSXProc xprocDDS = xprocService.newXProc(user, readOnly);

// Get the XProc processor object and run an XProc pipeline
XProc xproc = xprocDDS.getXProc();
Pipeline pipeline = xproc.newPipeline(new XMLSource(...));
PipelineInput input = pipeline.newPipelineInput();
input.addInput("source", new XMLSource(...));
PipelineOutput output = xproc.run(pipeline, input);
List<XMLSource> result = output.getXMLSources("result");
...

// Close all sessions
Collection<Session> openSessions = xprocDDS.getOpenSessions();
for (Session session : openSessions) {
```

```
  session.commit();
}
```

# XProc Service Configuration

To use the XProc service, Applications must configure the service in the services configuration file. The format of the XProc service configuration entry is as follows:

```
<xprocservice>
  <type class="DDSServiceType">XPROC</type>
  <name>XProcService</name>

<className>com.emc.documentum.xml.dds.xproc.impl.XProcServiceImpl</className>

  (I/O configuration)?
  (XSL formatter configuration)?
</xprocservice>
```

The I/O and XSL formatter sections are optional. In most cases, the default XProc service configuration should be sufficient.

I/O configuration

In the I/O configuration section, it is possible to register custom resolver modules (used by the XProc engine while resolving resources) and writer modules (used by the XProc engine while writing resources). These modules will be registered in addition to the standard (built-in) resolver/writer modules.

The format of the I/O configuration entry is as follows:

```
<io>
  <resolverModules>
    (<string>class name</string>)*
  </resolverModules>
  <writerModules>
    (<string>class name</string>)*
  </writerModules>
</io>
```

The classes specified in the `<resolverModules>` and `<writerModules>` must extend classes `com.emc.documentum.xml.dds.xproc.AbstractDDSResolverModule` and `com.emc.documentum.xml.dds.xproc.AbstractDDSWriterModule`, respectively, and must provide a default (no-argument) constructor.

XSL formatter configuration

In the XSL formatter section of the configuration, additional properties for the XSL formatter can be specified.

**Note:**

Currently, only Apache FOP is supported.

See http://xmlgraphics.apache.org/fop for information on FOP and the syntax of the FOP configuration file.

The entry has the following format:

```
<xslFormatter>
  <fop>
    <configurationLocation>path</configurationLocation>
```

```
  </fop>
</xslFormatter>
```

The entry `<configurationLocation>` specifies the path to the FOP configuration file. During initialization of the XProc service, the configuration file will be looked up in the Java classpath; if not found, the system will try to interpret the path as a file system path.

In order to use non-standard fonts with FOP, the following steps must be performed on the server machine:

1. Install the fonts if they are not available by default.
2. Generate FOP font metrics file (see http://xmlgraphics.apache.org/fop for more information)
3. Create a FOP user configuration file with proper font-related configuration.
4. Update the services configuration file of the Application so that it points to the FOP configuration file.
5. Make sure that the XSL stylesheets used for generating the XSL-FO content use proper fonts.

The following configuration registers a custom resolver module that provides the XProc processor with the ability to access content using the FTP protocol. In addition to that, the FOP formatter will be using configuration file `fop-config.xml`.

```
<xprocservice>
  <type class="DDSServiceType">XPROC</type>
  <name>XProcService</name>

<className>com.emc.documentum.xml.dds.xproc.impl.XProcServiceImpl</className>

  <io>
    <resolverModules>
      <string>com.example.xproc.FTPURIResolver</string>
    </resolverModules>
  </io>
  <xslFormatter>
    <fop>
      <configurationLocation>fop-config.xml</configurationLocation>
    </fop>
  </xslFormatter>
</xprocservice>
```

# Logic Engine Service

The Logic Engine Service provides the functionality of the S1000D Process Data Module Logic Engine to Applications.

The Logic Engine Service will be started automatically on Application startup, provided it has been configured for the Application (see Application Configuration and XProc Service Configuration for more information).

**Concepts**

A DDS Logic Engine represents a wrapper object on top the Logic Engine, that provides necessary integration with DDS and its frameworks.

A Logic Engine instance is always bound to a particular User. All resource manipulations (both read and write) are performed using this user information.

For resolving Process Data Modules (and other content), the Logic Engine implementation uses so called Data Module Resolver.

A (running) instance of a Process Data Module is represented by its state. The Logic Engine implementation stores the state information persistently in a form of an XML document. The

component that is responsible for storing (and retrieving) the state information is called the State Manager.

To use the Logic Engine service, custom implementations of the data module resolver and the state manager must be registered with the Logic Engine (see Logic Engine Service Configuration ).

When referring to resources, the Logic Engine needs to be able to create Sessions for the Persistence framework. In order to facilitate that, each Logic Engine instance is assigned a Session Pool which it can use for creating new sessions and for reusing existing sessions.

**Interfaces**

The Logic Engine Service concepts have been codified into interfaces in the `com.emc.documentum.xml.dds.le` package.

**DDSLogicEngine**

The Logic Engine interface (`com.emc.documentum.xml.dds.le.DDSLogicEngine`) provides accessors to the following:

- Logic Engine: the Logic Engine processor itself. The Logic Engine provides APIs for instantiating new process instances, performing navigation operations, etc.
- Session pool: provides the data module resolver and state manager with the ability to create and reuse Sessions. The session pool also keeps track of all Sessions that have been created while interacting with the Logic Engine.
- Open sessions: the collection of all Sessions that have been created while interacting with the Logic Engine.

**DDSProcessDataModuleRenderer**
The Process Data Module Renderer interface (`com.emc.documentum.xml.dds.le.DDSProcessDataModuleRenderer`) provides functionality for rendering Process Data Modules into printable representations:

- render(): this method renders a Process Data Module using a specified output media type.

The Process Data Module Renderer interface provides accessors to the following:

- Session pool: the session pool provides the Process Data Module Renderer with the ability to create and reuse Sessions. The session pool also keeps track of all Sessions that have been created while using the Process Data Module Renderer.
- Open sessions: the collection of all Sessions that have been created while using the Process Data Module Renderer.

**LogicEngineService**

The LogicEngineService interface (`com.emc.documentum.xml.dds.le.LogicEngineService`) provides the following methods:

- newLogicEngine(): returns a new Logic Engine object that represents a DDS-aware wrapper on top of the Logic Engine itself.
- newProcessDataModuleRenderer(): returns a new Process Data Module object.

**AbstractDDSDataModuleResolver**
Abstract implementation of the Data Module Resolver interface (`com.emc.documentum.xml.dds.le.AbstractDDSDataModuleResolver`) that provides basic integration with DDS. Extend this class if you want to implement a custom data module resolver. The DDS-aware data module resolver provides accessors to the following:

- DDS application: the Application object.
- DDS user: the User using the Logic Engine.
- Session pool: the session pool associated with the Logic Engine.

**AbstractDDSStateManager**

Abstract implementation of the State Manager interface (`com.emc.documentum.xml.dds.le.AbstractDDSStateManager`) that provides basic integration with DDS. Extend this class if you want to implement a custom state manager. The DDS-aware state manager provides accessors to the following:

- DDS application: the Application object.
- DDS user: the User using the Logic Engine.
- Session pool: the session pool associated with the Logic Engine.

**Usage**

To use the LogicEngineService, a reference is obtained from the Application object:

```
// Get the Logic Engine service instance
LogicEngineService logicEngineService =
(LogicEngineService)application.getServiceManager().getService(DDSServiceType.LOGICENGINE);

User user = ...;

// Create a new DDSLogicEngine instance
DDSLogicEngine logicEngineDDS = logicEngineService.newLogicEngine(user);

// Get the Logic Engine object and start a process
LogicEngine logicEngine = logicEngineDDS.getLogicEngine();
DataModuleRef dmRef = ...;

ProcessView view = logicEngine.initProcess(dmRef, null);
...
view = logicEngine.currentProcessView(dmRef, null);
...
logicEngine.releaseProcess(dmRef, null);

...

// Close all sessions
Collection<Session> openSessions = logicEngineDDS.getOpenSessions();
for (Session session : openSessions) {
  session.commit();
}
```

The following example demonstrates the use of the Process Data Module Renderer:

```
// Get the Logic Engine service instance
LogicEngineService logicEngineService =
(LogicEngineService)application.getServiceManager().getService(DDSServiceType.LOGICENGINE);

User user = ...;

// Create a new DDSProcessDataModuleRenderer instance
DDSProcessDataModuleRenderer rendererDDS =
logicEngineService.newProcessDataModuleRenderer(user);

// Render a Process Data Module
DataModuleRef dmRef = ...;
OutputStream out = ...;

rendererDDS.render(dmRef, out, OutputType.PDF);

...

// Close all sessions
Collection<Session> openSessions = rendererDDS.getOpenSessions();
for (Session session : openSessions) {
  session.commit();
}
```

**Related links**

- Logic Engine Service Configuration
- GWT Logic Engine Service
- Application Configuration

# Logic Engine Service Configuration

To use the Logic Engine service, DDS applications must configure the service in their Services Configuration file:

```
<logicengineservice>
  <type class="DDSServiceType">LOGICENGINE</type>
  <name>LogicEngineService</name>

<className>com.emc.documentum.xml.dds.le.impl.LogicEngineServiceImpl</className>

  <dataModuleResolver>class name</dataModuleResolver>
  <stateManager>class name</stateManager>
  (<s1000DVersion>version</s1000DVersion>)?
</logicengineservice>
```

The `<dataModuleResolver>` entry specifies the data module resolver to be used with the Logic Engine. The class must extend the class `com.emc.documentum.xml.dds.le.AbstractDDSDataModuleResolver` and provide a default (no-argument) constructor.

The `<stateManager>` entry specifies the state manager to be used with the Logic Engine. The class must extend the class `com.emc.documentum.xml.dds.le.AbstractDDSStateManager` and provide a default (no-argument) constructor.

The entry `<s1000DVersion>` is optional and can be used for specifying the version of the S1000D schemas to use. If omitted, version '3.0' will be used by default.

**Usage**

The following configuration registers class `com.example.le.Resolver` as the data module resolver and class `com.example.le.StateManager` as the state manager. S1000D version '2.3' will be used.

```
<logicengineservice>
  <type class="DDSServiceType">LOGICENGINE</type>
  <name>LogicEngineService</name>

<className>com.emc.documentum.xml.dds.le.impl.LogicEngineServiceImpl</className>

  <dataModuleResolver>com.example.le.Resolver</dataModuleResolver>
  <stateManager>com.example.le.StateManager</stateManager>
  <s1000DVersion>2.3</s1000DVersion>
</logicengineservice>
```

# DDS URIs

Applications refer to resources using URIs. DDS provides an extensible URI scheme as a unified way of representing and identifying resources.

This URI scheme is intended to:

- hide complexity of underlying Stores and their internal structure
- simplify the identification of resources for Developers
- loosen the coupling between client-side code and the actual data layout in the Store

**Concepts**
The syntax of DDS-specific URIs follows the generic URI syntax as defined in Internet standard STD 66 (or RFC3986 ). In its general form, a DDS URI has the following structure:

```
dds:[//<domain descriptor>]<domain specific part>
```

The domain descriptor, which is a registry-based authority string, carries information about the target domain of the URI. This information is described using a set of so called domain attributes. The domain descriptor specifies a (posibbly empty) sequence of attribute/value pairs, separated by a semicolon (';'). The structure of the attribute/value pair is:

```
<attr>=<value>
```

The names of attributes, their possible values, and their semantics are implementation-dependent.

The domain specific part: identifies the resource within the context of the target domain represented by the domain descriptor. Usually, it contains a path to the resource within the target domain. It can also contain an optional query and fragment identifier:

```
<path>[?<query>][#<frag>]
```

The URIs are hierarchical and can be relative or absolute. In general, a URI is said to be absolute if it contains a URI scheme (`dds`); if the scheme is not present, the URI is said to be relative.

A URI target represents the target object identified by a DDS URI.

A URI resolver resolves URIs to URI targets and provides functionality for generating URIs for existing content.

Applications have access to a URI Resolver implementation that is registered in the application bootstrap. A default URI resolver is provided. Applications can register custom URI resolvers.

Applications can use custom URIResolver implementations to provide

- for custom domain attributes
- custom URI resolution logic

URIs can be extended by introducing:

- new domain attributes
- new domain attributes values
- new domain specific components

**Default URI Resolver**
The default URI resolver defines the following optional domain attributes:

- STORE: Identifies the target domain Store by specifying the Store alias. If unspecified, the default Store of the application will be used. If no default Store exists, the URI is potentially invalid and using it may lead to errors.
- DOMAIN: Identifies the target domain type. Possible values:

    - data: The target domain is the data collection of an application data set. This is the default value.
    - resource: The target domain is the application resources collection.
    - user: The target domain is the user's personal folder.
- DATASET: Specifies the target domain data set, using the dataset alias. Only used if DOMAIN is 'data'. If not specified and DOMAIN is 'data', the default data set of the application is used. If there is no default data set, the URI is potentially invalid and may cause errors.
- LOCALE: Specifies the target domain locale. Only used if DOMAIN is 'data'. If not specified and DOMAIN is 'data', the default locale of the data set will be used. If the data set is locale-aware, but the default locale cannot be determined, the URI is potentially invalid and may cause errors.

The default URI resolver follows the convention that URIs that represent directory-like objects should end with a trailing slash. So, when referring to Locations, the URI must end with a slash; if it does not, it will be interpreted as a Container. The following URIs will therefore resolve to different objects:

```
dds:/dir1/dir2/
dds:/dir1/dir2
```

**Interfaces**

The core URI functionality concepts have been codified into interfaces and classes in the package `com.emc.documentum.xml.dds.uri`:

- `com.emc.documentum.xml.dds.uri.DDSURI` - This class represents a URI. It contains functionality for creating new URIs as well as for extracting information from existing URIs (domain attributes, domain specific part etc.)
- `com.emc.documentum.xml.dds.uri.URIResolver` - This interface represents a URI resolver. It contains methods for resolving URIs to URI targets, and methods for generating URIs from existing content. Custom implementations of this interface can be used with Applications.
- `com.emc.documentum.xml.dds.uri.URITarget` - This interface represents the URI target object that is the result of resolving a URI.

The package `com.emc.documentum.xml.dds.uri.resolver` contains the following URI resolver implementations:

- `DDURIResolver` - The default URI resolver implementation.
- `AbsoluteDomainResolver` - This URI resolver supports interpreting URIs as absolute paths in the target Store.

On the GWT client side, the following classes are available:

- `com.emc.documentum.xml.gwt.dds.util.DDSURI` - GWT representation of an DDSURI. This class provides the similar functionality to `com.emc.documentum.xml.dds.uri.DDSURI`.

Some of the DDSURI functionality is also available for use in XQueries. The following XQuery extension function from the DDS namespace (`http://www.emc.com/documentum/xml/dds`) can be used:

- `generate-uri($node as node()) as xs:string` - Generates (using the application-specifc URI resolver implementation) an DDSURI for given XML node.

**Usage**

The examples below assume an Application with the following specifications:

- a single store with alias `xmlstore`
- two data sets, one with alias `dataset1` (default) and one with alias `dataset2`.
- `dataset1` contains two locales: `en_US` (default), and `cs_CZ`.
- `dataset2` contains no locale information
- The default URI resolver implementation is used

A resource in the default data set, default locale:

```
dds:/tasks/task001.xml
```

A resource in the default dataset, locale `cs_CZ`:

```
dds://LOCALE=cs_CZ/tasks/task001.xml
```

A resource in `dataset2`:

```
dds://DATASET=dataset2/chapters/chapter7.xml
```

A resource in the default dataset, default locale, applying an XPointer expression:

```
dds:/tasks/task001.xml#xpointer(node()[2])
```

Referring to a Location resource in the default data set:

```
dds:/chapters/
```

Referring to an application resource:

```
dds://DOMAIN=resource/xproc/xsl-transform.xpl
```

A resource in the application user's personal folder:

```
dds://DOMAIN=user/preferences.xml
```

Generating URIs in an XQuery:

```
declare namespace dds='http://www.emc.com/documentum/xml/dds';
let $node := ... (: get the node :)
return dds:generate-uri($node)
```

For more examples, see the DDS demo applications.

# Client Services

## GWT Services

The GWT Services framework provides access to the functionality of the Service Framework through asynchronous remote procedure calls (RPC).

The GWT Client API uses the GWT Services layer. GWT Services are built on top of the Service Framework (see Services Framework ). They include:

- Application service
- I18N service
- Index service
- Log Center service
- Logic Engine service
- Persistence service
- Resource service
- User service
- XQuery service
- XML Persistence service
- XProc service

See the DDS JavaDoc documentation for a full overview.

**Note:** The examples below do not cover all methods or functionality of every service.

**Interfaces**

GWT Services have been codified into interfaces in the `com.emc.documentum.xml.dds.gwt.client.rpc` package.

The Services utility class (`com.emc.documentum.xml.dds.gwt.client.rpc.DDSServices`) can be used to easily get a remote service.

**Application service**

The Application service interface (`com.emc.documentum.xml.dds.gwt.client.rpc.application.ApplicationService`) provides a GWT RPC interface for application-related functionality.

**Application service example**

```
ApplicationServiceAsync applicationService =
DDSServices.getApplicationService();
applicationService.getApplicationContext(
  new AsyncCallback<SerializableApplicationContext>() {
    public void onSuccess(SerializableApplicationContext result) {
      // Handle the SerializableApplicationContext result here.
    }
    public void onFailure(Throwable caught) {
      // Handle exceptions here.
```

```
    }
});
```

**I18N service**

The I18N service interface (`com.emc.documentum.xml.dds.gwt.client.rpc.I18NService`) provides a GWT RPC interface for internationalization (I18N) related functionality.

**I18N service example**

```
I18NServiceAsync i18NService = DDSServices.getI18NService();
i18NService.getISO3Languages("eng",
  new AsyncCallback<Map<String, String>>() {
    public void onSuccess(Map<String, String> result result) {
      // Handle the map of language codes and language names here.
    }
    public void onFailure(Throwable caught) {
      // Handle exceptions here.
    }
});
```

**Index service**

The Index service interface (`com.emc.documentum.xml.dds.gwt.client.rpc.xdb.index.IndexService`) provides a GWT RPC interface for index related functionality.

**Index service example**

```
IndexServiceAsync indexService = DDSServices.getIndexService();
indexService.getIndexList(path,
  new AsyncCallback<List<SerializableXDBIndex>>() {
    public void onSuccess(List<SerializableXDBIndex> result) {
      // Handle the list of SerializableXDBIndexIf objects here.
    }
    public void onFailure(Throwable caught) {
      // Handle exceptions here.
    }
});
```

**Log Center service**

The Log Center service interface (`com.emc.documentum.xml.dds.gwt.client.rpc.LogCenterService`) provides a GWT RPC interface for logging messages from the client back to the server.

**Log Center service example**

```
try {
// Execute some code
} catch (RuntimeException caught) {
  LogCenterServiceAsync logCenterService =
DDSServices.getLogCenterService();
  logCenterService.exception(getClass().getName(), caught,
    new AsyncCallback<Object>() {
      public void onSuccess(Object result) {
        // Void method, so the returned result will be null.
      }
      public void onFailure(Throwable caught) {
        // Handle exceptions here.
```

```
      }
  });
}
```

**LogicEngine service**

The LogicEngine service interface (`com.emc.documentum.xml.dds.gwt.client.rpc.le.LogicEngineService`) provides a GWT RPC interface for interacting with the S1000D Logic Engine.

**LogicEngine service example**

```
LogicEngineServiceAsync logicEngineService =
DDSServices.getLogicEngineService();
SerializableDataModuleRef dataModuleReference = new
SerializableDataModuleRef(documentURI, true);
logicEngineService.initProcess(dataModuleReference, null,
  new AsyncCallback<SerializableProcessView>() {
    public void onSuccess(SerializableProcessView result) {
      // Handle the SerializableProcessView result here.
    }
    public void onFailure(Throwable caught) {
      // Handle exceptions here.
    }
});
```

**Persistence service**

The Persistence service interface (`com.emc.documentum.xml.dds.gwt.client.rpc.persistence.PersistenceService`) provides a GWT RPC interface for persistence related functionality.

**Persistence service example**

```
PersistenceServiceAsync persistenceService =
DDSServices.getPersistenceService();
persistenceService.copy(sourceURI, targetURI,
  new AsyncCallback<Long>() {
    public void onSuccess(Long result) {
      // Handle the Long result here.
    }
    public void onFailure(Throwable caught) {
      // Handle exceptions here.
    }
});
```

**Resource service**

The Resource service interface (`com.emc.documentum.xml.dds.gwt.client.rpc.persistence.ResourceService`) provides a GWT RPC interface for retrieving resources from the server-side class path.

A resource is some data (text, xml, etc.) that can be accessed by class code in a way that is independent of the location of the code. The resources are referred to using '/'-separated path names, such as "com/example/resources/data.xml".

**Resource service example**

```
ResourceServiceAsync resourceService = DDSServices.getResourceService();
nameArray = new String[] { "com/example/resources/data.xml",
"com/example/resources/data2.xml" };
```

```
resourceService.getResourcesAsString(nameArray,
  new AsyncCallback<List<String>>() {
    public void onSuccess(List<String> result) {
      // Handle the list of strings here.
    }
    public void onFailure(Throwable caught) {
      // Handle exceptions here.
    }
});
```

**User service**

The User service interface (`com.emc.documentum.xml.dds.gwt.client.rpc.application.UserService`) provides a GWT RPC interface for resource-related functionality.

**User service example**

```
UserServiceAsync userService = DDSServices.getUserService();
userService.login(username, password,
  new AsyncCallback<Boolean>() {
    public void onSuccess(Boolean result) {
      // Handle the Boolean result here.
    }
    public void onFailure(Throwable caught) {
      // Handle exceptions here.
    }
});
```

**XQuery service**

The XQuery service interface (`com.emc.documentum.xml.dds.gwt.client.rpc.xquery.XQueryService`) provides a GWT RPC interface for XQuery execution.

**XQuery service example**

```
XQueryServiceAsync xqueryService = DDSServices.getXQueryService();
xqueryService.execute(uri, ".//topicref/@href"
  new AsyncCallback<List<SerializableXQueryValue>>() {
    public void onSuccess(List<SerializableXQueryValue> result) {
      // Handle the list of SerializableXQueryValue objects here.
    }
    public void onFailure(Throwable caught) {
      // Handle exceptions here.
    }
});
```

**XML Persistence service**

The XML Persistence service interface (`com.emc.documentum.xml.dds.gwt.client.rpc.persistence.XMLPersistenceService`) provides a GWT RPC interface for XML persistence functionality.

**XML Persistence service example**

```
DDSURI resourceURI = new DDSURI("temp/");
resourceURI.setAttribute(DDSURI.ATTRIBUTE_DOMAIN, DDSURI.DOMAIN_RESOURCE);
XMLPersistenceServiceAsync xmlService =
DDSServices.getXMLPersistenceService();
xmlService.getChildren(resourceURI.toString(),
```

```
SerializableNode.DOCUMENT_NODE,
  new AsyncCallback<List<SerializableNode>>() {
    public void onSuccess(List<SerializableNode> result) {
      // Handle the list of SerializableNode objects here.
    }
    public void onFailure(Throwable caught) {
      // Handle exceptions here.
    }
});
```

**XProc service**

The XProc service interface (`com.emc.documentum.xml.dds.gwt.client.rpc.xproc.XProcService`) provides a GWT RPC interface for running XProc pipelines.

**XProc service example**

```
XProcServiceAsync xprocService = DDSServices.getXProcService();

SerializablePipelineInput input = new SerializablePipelineInput();
input.addInput("source", new SerializableXMLSource(uri));

xprocService.runPipeline("classpath:resources/xproc/xml-pipe.xpl", input,
 true,
  new AsyncCallback<SerializablePipelineOutput>() {
    public void onSuccess(SerializablePipelineOutput result) {
      // Handle the SerializablePipelineOutput here.
    }
    public void onFailure(Throwable caught) {
      // Handle exceptions here.
    }
});
```

**Related links**

- GWT Application Service
- GWT User Service
- GWT Log Center Service
- GWT Persistence Service
- GWT XML Persistence Service
- GWT Resource Service
- GWT Index Service
- GWT XQuery Service
- GWT I18N Service
- GWT XProc Service
- GWT Logic Engine Service

# GWT Widget Library

DDS supports use of the Google Web Toolkit (GWT) for client-side development. The DDS GWT Widget Library provides application developers with a collection of GWT-based browser widgets and associated services, tailored for interaction with xDB.

**Concepts**

The GWT Widget Library allows application developers to abstract the DOM based elements in the browser into functional components. Written in Java, the GWT widgets manipulate DOM elements to execute the actual functionality. Interaction between components is further abstracted by event listeners, so components need not communicate with each other directly: widgets can fire and catch events, and strong links between widgets are not required. For information on events and listeners, go to the GWT website, under Docs, User Interfaces, Events and Listeners.

GWT supplies internationalization mechanisms, so that messages, labels etc. can easily be translated.

All pure GWT widget functionality resides on the client side. Interaction with a server is accomplished through remote procedure calls and HTTP requests. Scalability depends primarily on server-side considerations like the amount of data and interaction with remote servers

Customization or extension can be achieved by extending the widget classes or by modifying the CSS (Cascading Style Sheets).

Specific widgets exist to supply higher level and more specialized functionality. For example, there are widgets specifically designed to ease interaction with xDB and with the Logic Engine, for trees based on XQuery, for content rendering, as well as basic components such as spinners, date pickers etc.

The core widgets in the `com.emc.documentum.xml.gwt` packages are independent from other DDS components.

**Usage**

For usage examples, see the kitchensink demo application and other demos.

**Note:**
The Google Web Toolkit should be present on the development system. Specific versions exists for Windows, Linux and MacOS X. When developing with the GWT widget library, gwt-user.jar and gwt-dev-<operatingsystem>.jar must be referenced.

# GWT Application Service

The GWT Application Service allows developers to retrieve configuration information about the DDS Application.

This information is made available by means of an Application Context object.

This is codified in the `com.emc.documentum.xml.dds.gwt.client.rpc.ApplicationService` interface, which provides the following method:

- getApplicationContext(): returns the application context object that provides access to configuration information about the DDS application.

**Usage**
The following example demonstrates the usage of the Application Service in a GWT client application:

```
ApplicationServiceAsync applicationService =
DDSServices.getApplicationService();
applicationService.getApplicationContext(
  new AsyncCallback<SerializableApplicationContext>() {
    public void onSuccess(SerializableApplicationContext result) {
      // Handle the SerializableApplicationContext result here.
    }
    public void onFailure(Throwable caught) {
      // Handle exceptions here.
```

```
    }
});
```

**Related links**

- GWT Services
- Application Framework

# GWT User Service

The GWT User Service provides Developers a GWT front-end for the User Service.

This is codified in the `com.emc.documentum.xml.dds.gwt.client.rpc.application.UserService` interface, which provides the following methods:

- createUser(): for creation of Users.
- deleteUser(): for deletion of Users.
- login(): for logging in of a User.
- logout(): for logging out of a User.
- existsUser(): check for the existence of a User.
- storeUserObject() allows the User to store objects at its Home Location. The supplied path is relative to the Home Location
- retrieveUserObject() allows the User to retrieve objects from its Home Location. The supplied path is relative to the Home Location

**Related links**

- User Service

# GWT Log Center Service

The GWT Log Center Service provides the developers with the ability log messages back to the server.

This is codified in the `com.emc.documentum.xml.dds.gwt.client.rpc.LogCenterService` interface, which provides the following methods:

- log(): logs a message.
- warning(): logs a warning message.
- error(): logs an error message.
- exception(): logs an exception message.

The following example demonstrates the usage of the Log Center Service in a GWT client application:

```
try {
  // Execute some code
} catch (RuntimeException caught) {
  LogCenterServiceAsync logCenterService =
DDSServices.getLogCenterService();
  logCenterService.exception(getClass().getName(), caught,
    new AsyncCallback<Object>() {
      public void onSuccess(Object result) {
        // Void method, so the returned result will be null.
      }
```

```
      public void onFailure(Throwable caught) {
        // Handle exceptions here.
      }
  });
}
```

**Related links**

- GWT Services
- Logging Framework

# GWT Persistence Service

The GWT Persistence Service provides the DDS application developers with the ability to access the functionality of the persistence service in GWT client code.

This is codified in the `com.emc.documentum.xml.dds.gwt.client.rpc.persistence.PersistenceService` interface, which provides the following methods:

- createDocument(): creates an XML document with given URI.
- createLibrary(): creates a location with given URI.
- createLibraries(): creates locations with given URIs.
- move(): moves a store child.
- copy(): copies a store child.
- remove(): removes a store child.

The following example demonstrates the usage of the Persistence Service in a GWT client application:

```
PersistenceServiceAsync persistenceService =
DDSServices.getPersistenceService();
persistenceService.copy(sourceURI, targetURI,
  new AsyncCallback<Long>() {
    public void onSuccess(Long result) {
      // Handle the Long result here.
    }
    public void onFailure(Throwable caught) {
      // Handle exceptions here.
    }
});
```

**Related links**

- GWT Services
- Persistence Layer

# GWT XML Persistence Service

The GWT Persistence Service provides the DDS application developers with the ability to access the XML-related functionality of the persistence service in GWT client code.

This is codified in the `com.emc.documentum.xml.dds.gwt.client.rpc.persistence.XMLPersistenceService` interface, which provides the following methods:

- getNode(): returns the node with given URI.
- getNodes(): returns a list of nodes with given URIs.

- getChildCount(): returns the number of child nodes of node with given URI.
- getChildren: returns the child nodes of node with given URI.
- getContentAsString(): returns the content of node with given URI as string.
- insert(): inserts an XML fragment.
- move(): moves an XML fragment.
- copy(): copies an XML fragment.
- remove(): removes a node with given URI.
- setAttribute(): sets an attribute on element with given URI.
- setAttributes(): sets attributes on element with given URI.

The following example demonstrates the usage of the XML Persistence Service in a GWT client application:

```
String resourceURI = ...;
XMLPersistenceServiceAsync xmlService =
DDSServices.getXMLPersistenceService();
xmlService.getChildren(resourceURI, SerializableNode.DOCUMENT_NODE,
  new AsyncCallback<List<SerializableNode>>() {
    public void onSuccess(List<SerializableNode> result) {
      // Handle the list of SerializableNode objects here.
    }
    public void onFailure(Throwable caught) {
      // Handle exceptions here.
    }
});
```

**Related links**

- GWT Services
- Persistence Layer

# GWT Resource Service

The GWT Resource Service provides Developers with the ability to resolve resources from the server-side classpath.

Classpath resources are referred to using '/'-separated path names such as "com/example/resources/data.xml".

This is codified in the `com.emc.documentum.xml.gwt.client.rpc.ResourceService` interface, which provides the following methods:

- getResourceAsString(): returns the specified classpath resource and returns its content as string.
- getResourcesAsString(): returns the specified classpath resources and returns their content as a string array.

The following example demonstrates the usage of the Resource Service in a GWT client application:

```
ResourceServiceAsync resourceService = DDSServices.getResourceService();
nameArray = new String[] { "com/example/resources/data.xml",
"com/example/resources/data2.xml" };
resourceService.getResourcesAsString(nameArray,
  new AsyncCallback<List<String>>() {
    public void onSuccess(List<String> result) {
      // Handle the list of strings here.
    }
    public void onFailure(Throwable caught) {
      // Handle exceptions here.
    }
});
```

**Related links**

- GWT Services

# GWT Index Service

The GWT Index Service provides developers with the ability to manipulate xDB indexes.

This is codified in the `com.emc.documentum.xml.dds.gwt.client.rpc.xdb.index.IndexService` interface, which provides the following methods:

- getIndexList(): returns the list of indexes for given target object.
- getKeys(): returns the keys in the index.
- getStoragePages(): returns the number of storage pages used by an index.
- removeIndexes(): removes indexes from the list of indexes of given target object.

The following example demonstrates the usage of the Index Service in a GWT client application:

```
IndexServiceAsync indexService = DDSServices.getIndexService();
indexService.indexService.getIndexList(path,
  new AsyncCallback<List<SerializableIndex>>() {
    public void onSuccess(List<SerializableIndex> result) {
      // Handle the list of SerializableIndex objects here.
    }
    public void onFailure(Throwable caught) {
      // Handle exceptions here.
    }
});
```

**Related links**

- GWT Services

# GWT XQuery Service

The GWT XQuery Service allows developers to execute XQueries on the server side.

This is codified in the `com.emc.documentum.xml.dds.gwt.client.rpc.xquery.XQueryService` interface, which provides the following method:

- execute(): executes an XQuery on a context object specified by a URI. Optionally, it is possible to specify values for external variables used in the XQuery.

The following example demonstrates the usage of the XQuery Service in a GWT client application:

```
XQueryServiceAsync xqueryService = DDSServices.getXQueryService();
xqueryService.execute(uri, ".//topicref/@href"
  new AsyncCallback<List<SerializableXDBXQueryValueIf>>() {
    public void onSuccess(List<SerializableXDBXQueryValueIf> result) {
      // Handle the list of SerializableXDBXQueryValueIf objects here.
    }
    public void onFailure(Throwable caught) {
      // Handle exceptions here.
    }
});
```

**Related links**

- GWT Services

# GWT I18N Service

The GWT I18N Service provides internationalization (I18N) related functionality in the client applications.

This is codified in the `com.emc.documentum.xml.dds.gwt.client.rpc.I18NService` interface, which provides the following methods:

- getCountries(): returns all defined ISO country codes and names.
- getCountryName(): returns the country name for given ISO country code.
- getLanguages(): returns all defined ISO language codes and languages
- getLanguageName(): returns the language name for given ISO language code.
- getISO2Countries(): returns all defined ISO 2-character country codes and country names.
- getISO2CountryCodes(): returns all defined ISO 3-character country codes.
- getISO2Languages(): returns all defined ISO 2-character language codes and language names.
- getISO2LanguageCodes(): returns all defined ISO 3-character language codes.
- getISO3Countries(): returns all defined ISO 3-character country codes and country names.
- getISO3CountryCodes(): returns all defined ISO 3-character country codes.
- getISO3Languages(): returns all defined ISO 3-character language codes and language names.
- getISO3LanguageCodes(): returns all defined ISO 3-character language codes.

The following example demonstrates the usage of the I18N Service in a GWT client application:

```
I18NServiceAsync i18NService = DDSServices.getI18NService();
i18NService.getISO3Languages("eng",
  new AsyncCallback<Map<String, String>>() {
    public void onSuccess(Map<String, String> result result) {
      // Handle the map of language codes and language names here.
    }
    public void onFailure(Throwable caught) {
      // Handle exceptions here.
    }
});
```

**Related links**

- GWT Services

# GWT XProc Service

Developers can use the GWT XProc Service for a GWT front-end for the XProc Service.

This is codified in the `com.emc.documentum.xml.dds.gwt.client.rpc.xproc.XProcService` interface, which provides the following method:

- runPipeline(): executes an XProc pipeline on the server and returns the result. The pipeline is identified by a URI.

The GWT XProc Service uses the following GWT-specific classes in the `com.emc.documentum.xml.gwt.xproc.client.rpc` package:

- `SerializableQName`: GWT equivalent of the standard Java class `java.xml.namespace.QName`. Used for representing XML qualified names.
- `SerializableXMLSource`: GWT equivalent of the class `com.emc.documentum.xml.xproc.io.XMLSource` which represents XML content that can be processed (or produced) by the XProc engine.
- `SerializablePipelineInput`: GWT equivalent of the class `com.emc.documentum.xml.xproc.pipeline.model.PipelineInput` which represents input to an XProc pipeline.
- `SerializablePipelineOutput`: GWT equivalent of the class `com.emc.documentum.xml.xproc.pipeline.model.PipelineOutput` which represents output of an XProc pipeline.

**Usage**
The following example demonstrates usage of the XProc Service in a GWT client application.

```
String pipelineURI = ...;
String dataURI = ...;
boolean readOnly = true;

// Get the GWT XProcService instance
XProcServiceAsync xprocService = DDSServices.getXProcService();

// Prepare the input data to the pipeline
SerializablePipelineInput input = new SerializablePipelineInput();
input.addInput("source", new SerializableXMLSource(dataURI));
input.setOption(new SerializableQName("output-format"), "xhtml");

// Run the pipeline
xprocService.runPipeline(pipelineURI, input, readOnly,

    new AsyncCallback<SerializablePipelineOutput>() {

        // Process pipeline result
        public void onSuccess(final SerializablePipelineOutput output) {
          List<SerializableXMLSource> sources =
output.getXMLSources("result");
          for (SerializableXMLSource source : sources) {
            //Process each SerializableXMLSource object
            ...
          }
        }

        // Handle errors encountered during pipeline execution
        public void onFailure(final Throwable caught) {
          ...
        }
    }
);
```

# GWT Logic Engine Service

Developers can use the GWT Logic Engine Service for a GWT front-end for the Logic Engine Service.

This is codified in the `com.emc.documentum.xml.dds.gwt.client.rpc.le.LogicEngineService` interface, which provides the following methods:

- initProcess(): starts a new Process Data Module instance.
- releaseProcess(): aborts a running Process Data Module instance.
- processExists(): checks whether an instance of a Process Data Module exists.

- isProcessFinished(): checks whether a Process Data Module instance has reached the final process state.
- currentProcessView(): returns the current view on a Process Data Module Instance.
- previous(): navigates to the previous step of a Process Data Module instance and returns the new process view.
- next(): navigates to the next step of a Process Data Module instance and returns the new process view.
- cancelDialog(): cancels a dialog of a Process Data Module instance and returns the new process view.
- submitDialog(): submits a dialog of a Process Data Module instance and returns the new process view.
- setFillinValue(): sets a value for a 'fill-in' entry in a dialog of a Process Data Module instance and returns the new process view.
- setMenuChoiceSelected(): sets the state of a menu choice in a dialog of a Process Data Module instance and returns the new process view.
- pushButton(): presses a button in a dialog of a Process Data Module instance and returns the new process view.
- submitExtAppResponse: submits results of running an external application for a Process Data Module instance and returns the new process view.

These methods always expect the following two parameters:

- data module reference: reference to a Process Data Module.
- process owner data: additional data to be associated with the current User. This information can be used by the server-side State Manager implementation for more fine-grained identification of running Process Data Module instances. This parameter can be left unspecified.

The Logic Engine Service uses the following GWT-specific interface:

- `com.emc.documentum.xml.le.gwt.client.rpc.SerializableProcessOwner`: implementations of this interface can be used for specifying additional information associated with the current User. Server-side State Manager implementation can use this information for identification of running Process Data Module instances.

The Logic Engine Service uses the following GWT-specific classes in the `com.emc.documentum.xml.le.gwt.client.rpc` package:

- `SerializableDataModuleRef`: GWT equivalent of the class `com.emc.documentum.xml.le.engine.resolver.DataModuleRef`: represents a Data Module reference.
- `SerializableProcessView`: represents an abstract view on a Process Data Module instance.
- `SerializableBaselineProcessView`: represents a process view that supports navigating to the previous or next step of a Process Data Module instance.
- `SerializableDialogProcessView`: represents a process view with a contained dialog.
- `SerializableDataModuleProcessView`: represents a process view with a contained Data Module.
- `SerializableStepsProcessView`: represents a process view with a contained dialog.
- `SerializableExtAppProcessView`: represents a process view that should be used for executing an external application.
- `SerializableTerminalProcessView`: represents the terminal state of a Process Data Module instance.
- `SerializableDialog`: represents a dialog.
- `SerializableDialogItem`: represents a dialog item.
- `SerializableDialogGroup`: represents a group of dialog items.
- `SerializableFillin`: represents the 'fill-in' dialog item.
- `SerializableMenu`: represents a menu dialog item.
- `SerializableMenuChoice`: represents amenu choice.

- `SerializableMessage`: represents a message dialog item.
- `SerializableValidationStatus`: represents the validation status of a dialog item.
- `SerializableExtApp`: provides access to the information needed to execute an external application.
- `SerializableExtAppResponse`: represents the result of running an external application.
- `SerializableNamedValue`: a name/value pair used for representing parameters passed to and values returned by an external application.

**Usage**

The following example demonstrates how to start a new Process Data Module instance GWT client application.

```
SerializableDataModuleRef dmRef = ...;

// Get GWT LogicEngineService instance
LogicEngineServiceAsync leService = DDSServices.getLogicEngineService();

// Start a new process
leService.initProcess(dmRef, null,
    new AsyncCallback<SerializableProcessView>() {

        // Process (display) the returned view object
        public void onSuccess(SerializableProcessView result) {
          ...
        }

        // Handle errors
        public void onFailure(Throwable caught) {
          ...
        }
    }
);
```

The following example demonstrates how to navigate to the next step of a Process Data Module instance.

```
SerializableDataModuleRef dmRef = ...;

// Get GWT LogicEngineService instance
LogicEngineServiceAsync leService = DDSServices.getLogicEngineService();

// Navigate to the next step of the process
leService.next(dmRef, null,
    new AsyncCallback<SerializableProcessView>() {

        // Process (display) the returned view object
        public void onSuccess(SerializableProcessView result) {
          ...
        }

        // Handle errors
        public void onFailure(Throwable caught) {
          ...
        }
    }
);
```

**Related links**

- Logic Engine Service

# DDS Tag Library

Developers can use a library of DDS-specific tags in JSP-based applications (as opposed to, for example, GWT-based applications).

Below is a list of supported tags with a description of each tag, its optional and required attributes, and the possible exceptions that may occur. The prefix used is arbitrary, as long as it is properly declared in the JSPs.

**<dds:session>**

Every time you use DDS tags in a JSP, you should use session tag as the outermost tag for proper session management. Nesting session tags is of no use.

*Optional*

**userName**

(Example: username="Administrator")
The username for the session.

**retries**

the maximum number of retries that should be attempted when getting deadlock exceptions

**readOnly**

Indicates whether the Sessions under this tag should be read-only or not.
(Example: readOnly="true")

**<dds:contextnode>**

This tag is used to set the context node, for example for XQueries or transformations (see below). The contextnode tag is normally used once before the outermost foreach tag (see below), to set the initial contextnode for XQueries.

*Required*

**uri**

(Example: uri =
"dds://DOMAIN=data;DATASET=garage;LOCALE=en_US/concepts/oil.xml")

**<dds:tostring>**

This tag copies the toString value of the current context node to the page output.

**<dds:transform>**

The transform tag is capable of tranforming the contents of a node into any other format by using an XSLT transformation. The XSLT source is found either directly (styleURI attribute) or through an XQuery (baseURI plus XQuery)

*Optional*

**uri**

(Example: uri =
"dds://DOMAIN=data;DATASET=garage;LOCALE=en_US/concepts/oil.xml").
The URI indicating the location under which the content to be transformed can be found. If omitted, the context node is used.

**styleURI**

(Example: styleURI="dds://DOMAIN=resource/xslt/taglib/taglibe-demo.xsl")

**baseURI**

(Example: baseURI="dds://DOMAIN=resource/xslt/taglib/")

**xquery**

(Example: xquery="for $x in document('taglib-demo.xsl') return $x")
styleURI takes precedence over baseURI/xquery.

*Exceptions*

If no transformable content or no transformation is found after all these options, a JspException is thrown.

**<dds:print>**

The print tag copies its body content directly to the page output.

**<dds:xquery>**

The XQuery tag performs an XQuery, specified in its body content, on the current context node (see above). The result is stored in the page context, to be used by other tags (when and foreach, see below).

*Optional*

**contextNodeURI**

(Example: contextNodeURI = "dds://DOMAIN=data;DATASET=garage;LOCALE=en_US/concepts/")
This attribute sets the context node for the query.

*Exceptions*

If no contextnode is found, either in the page context (example: as set by a previous contextnode tag), or by the contextNodePath attribute, a JspException results. Also, if the context node found is not a LibraryChild (see XML Store documentation), a JspException results.

**<dds:when>**

This tag is used in situations where an action (example: a transform or tostring) should be performed conditionally on the outcome of some action. This action might be an XQuery with a boolean result. Within a when tag, other tags are executed if and only if the XQuery result is true. Another possible action is a test-method on a user-specified class. This happens when the programmer sets the optional classname attribute (see below)

*Optional*

**className**

(Example: className="someFullySpecifiedClassName")
This attribute sets the className of the class where the tag will find the boolean "test" method, with a Node as its single parameter. If and only if the method returns true for the current context node, the tags within the when tag are executed.

*Exceptions*

If the xquery yields the wrong type of result, a JspException is thrown. If the "test" method is not a boolean method, or has the wrong parameters, or the className does not specify a valid class, a JspException results.

**<dds:choose>**

Multiple when tags can be grouped within a choose tag. The first when tag preceded by an XQuery resulting in true gets to have its body executed. All other tags are skipped.

**<dds:otherwise>**

If all when tags in a choose tag are skipped because none of the XQueries yields true, then the body of the otherwise tag is executed.

**<dds:foreach>**

This tag is used in situations where an action (example: a transform or tostring) should be performed for each result coming from the result set of a preceding XQuery. This XQuery must yield a nodeset result. In each iteration, the context node for the tags embodied by the foreach tag is set to a new node from this nodeset.

*Exceptions*

If there is any result other than a node in the result set, a JspException is thrown.

**<dds:break>**

The break tag is similar to the java break statement, interrupting the closest enclosing foreach tag.

**<dds:xform>**

The xform tag locates an XForm stored in a DDS Application, and displays it in the resulting view. A JSP using the XForm tag should include some additional content in the <head> section. See the samples on what exactly this should be.

*Optional*

**uri**
(Example: uri="dds://DOMAIN=resource/xforms/GarageTitleSearch/")
The URI indicating the location where the documents for the XForm can be found. See the Administration chapter for information on how XForms are stored. If omitted, the context node is used.

**locale**
(Example: locale="en_US")
Attribute indicating the locale for the form to select.

*Exceptions*
If no location is found under the URI, a JspException results. Also, if the location does not contain an XForm, a JspException results.

**<dds:xproc>**
The xproc tag executes an XProc pipeline stored in a DDS application. Nested within this tag can be input, option, and parameter tags (described below).

*Optional*

**uri**
(Example: uri="dds://DOMAIN=resource/xproc/dita-html.xpl")
The URI indicating the location where the pipeline can be found. If omitted, the context node is used.

**output**
(Example: output="out")
The output port specifies the name of the port whose output should be written to the JSP's output. If omitted, the pipeline's primary output port's output is written to the JSP's output.

**ignoreoutput**
(Example: ignoreoutput="false")
Boolean attribute specifying whether or not to ignore the pipeline's output altogether. Default is false.

**<dds:input>**
The input tag specifies an input to an XProc pipeline port. This tag is only meaningful within an xproc tag.

*Optional*

**uri**
(Example: dds://DOMAIN=data;DATASET=garage;LOCALE=eng/concepts/oil.xml")
The URI indicating the location where the port's input can be found. If omitted, the input tag's content is used as the port's input.

**port**
(Example: port="stylesheet")
The port through which the input should enter the pipeline. If omitted, the input enters through the pipeline's primary input port.

**<dds:option>**
The input tag specifies an option to an XProc pipeline. This tag is only meaningful within an xproc tag.

*Optional*

**prefix**
(Example: prefix="dds")

**namespaceURI**

(Example: namespaceuri="http://com.emc.documentum/xml/dds/taglib")

*Required*

**name**
(Example: name="show_xml_declaration")
The prefix, name, and namespaceuri together form a QName. Only name (representing the QName's local part) is required.

**value**
(Example: value="true") The value of the option.

**<dds:parameter>**

The parameter tag specifies a parameter to an XProc pipeline. This tag is only meaningful within an xproc tag.

*Optional*

**prefix**
(Example: prefix="dds")

**namespaceURI**
(Example: namespaceuri="http://com.emc.documentum/xml/dds/taglib")

*Required*

**name**
(Example: name="show_xml_declaration")
The prefix, name, and namespaceuri together form a QName. Only name (representing the QName's local part) is required.

**value**
(Example: value="true")
The value of the parameter.

**<dds:blob>**

The blob tag inserts an image element into the output with the proper URI to be able to locate the image.

*Optional*

**uri**
(Example: uri = "dds://DOMAIN=data;DATASET=garage;
LOCALE=eng/image/carwash.gif")
The URI pointing to the blob resource. The result in the output, based on the above example will be: <img src="Blobservlet?uri=dds://DOMAIN=data;DATASET=garage;
LOCALE=eng/image/carwash.gif"/>.

**alt**
(Example: alt="Carwash")
The value of the alt argument to include for the resulting img element.

**Usage**

The following example demonstrates usage of the Tag library in a JSP. First, the context node is set to the /concepts/oil.xml document in the English locale of the garage dataset. This document is then transformed using the stylesheet found under the styleURI of the transform tag.

```
<!doctype html public "-//w3c//dtd html 4.0 transitional//en">
<html>
<%@ taglib uri="http://www.emc.com/documentum/xml/dds/taglib" prefix="dds"
 %>

<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>DDS Taglib Demo</title>
</head>
```

```
<body>
  <h1>DDS Tag Libs Sample</h1>
  <dds:session>
    <dds:contextnode
uri="dds://DOMAIN=data;DATASET=garage;LOCALE=eng/concepts/oil.xml"/>
    <dds:transform
styleURI="dds://DOMAIN=resource/xslt/taglib/taglib-demo.xsl" />
</dds:session>
</body>
</html>
```

**Taglib API**

There are some convenience classes in the `com.emc.documentum.xml.dds.taglib` package in the API for adding your own tags. The `TagHelper` class contains some DDS-specific convenience methods. For the tag classes themselves, we recommend extending `AbstractTag` or `AbstractBodyTag` and implementing the `performStartTag()` method. This prevents the tag from executing if an exception was thrown earlier. See the JavaDocs on the `com.emc.documentum.xml.dds.taglib` package for a more complete description.

To ensure that your tags (definitions, classes, attributes) will be found and interpreted correctly by the web container, you must include your custom tags in a .tld file, and refer to this .tld file from the application's web.xml file.

# DDS Admin Client

## The Admin user interface

The Admin application provides views on applications, and on data sets and their indexes. You can access these views by clicking the corresponding tab in the tab menu bar:

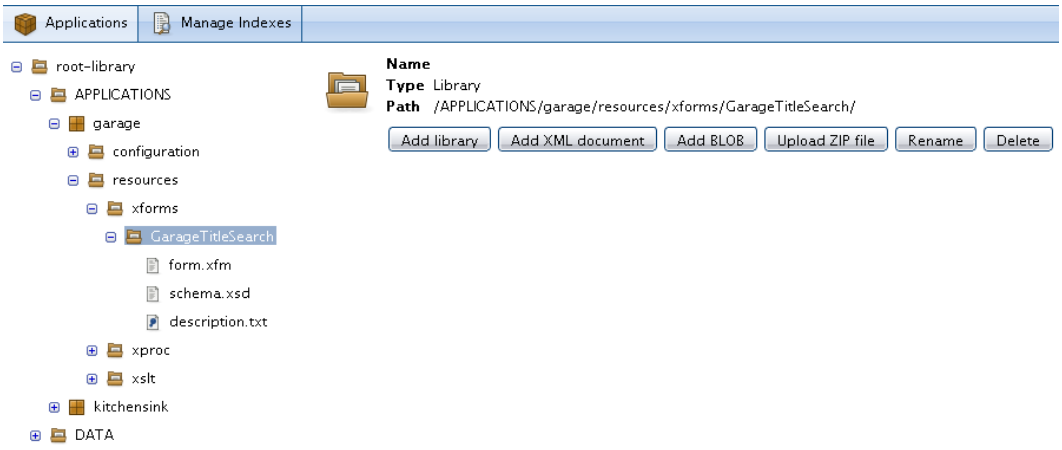
Applications | Manage indexes

The Admin views have a common layout, which includes (from top to bottom):

- Logo area
- Views Tab area
- View area
  - Navigation pane (left)
  - Info pane (top right: icon, Name, Type, Path of selected item)
  - Content pane (context-sensitive)
    - on **Applications** tab, shows buttons for the selected object
    - on **Manage Indexes** tab, shows tabs and buttons for the selected object
- Version area

**Applications, libraries and documents**

Applications can be created only in the root folder of the APPLICATIONS tree. A library can be added to an application or to another library. A document can be added to an application or to a library.




**Indexes**

Index Manager hides much of the complexity of generating indexes for query optimization.

**Icons in the Admin interface**

Admin indicates object types visually by means of icons:

| Icon | Object type |
|---|---|
| | Application |
| | Library |
| | XML Document |
| | Non-XML document (blob) |

# Navigation

Admin manages a repository where you store applications and the content they use. Your organization might use multiple repositories. Each repository is comprised of nodes that give access to the repository's content and functions. Admin displays the repository's nodes in a tree view.

1. Select a menu tab.

   By default, the navigation pane appears with the top item in the tree selected, and displaying the next level of nodes.

2. To navigate the repository tree, do any of the following.
   - To expand a collapsed node, click its plus icon.
   - To collapse an expanded node, click its minus icon.

   **Note:** Once you have clicked in the navigation pane, you can also use the Up and Down cursor control keys on your keyboard for navigation, and the Left and Right cursor control keys to collapse and expand nodes.

3. To select a node, click its name.

   The corresponding library or document appears in the Info pane.

# Using dialogs

If Admin requires additional information for executing a function, for example for adding a document to a library, it will present a dialog in a separate window. The Admin window itself is inactivated until you close the dialog. Dialogs have a number of properties in common, including:

- A title bar at the top, containing :
  - Title of the dialog.
  - path of the node that the function applies to.
  - A close button (X). Click this button to cancel the function and close the dialog.
- A work area, containing:
  - One or more data entry fields, usually with descriptive labels in front, and sometimes with action buttons, for example to **Browse** to file location.

- A **Submit** or similar button to execute the function and close the dialog.

For some functions, Admin evaluates or checks your input, which may result in a warning or error message.

1. Enter data.

   **Note:** On some dialogs, you can use Tab and Shift+Tab to move the text cursor from field to field.

2. Activate/submit or cancel the dialog.
   - To execute, click its **Submit** button at the bottom of the dialog.
   - To cancel, click its X button at the top of the dialog.

# Applications and Data

As was mentioned in the Administration section, DDS database stores two kinds of data:

- One or more data sets: each data set is a collection of actual (publishable) content data and its attendant metadata.
- Application data: data that is internal to the web applications, for example scripts of XQueries, stylesheets, XProcs, XForms etc.

**Application data**

Application data is organized into libraries. A library can contain zero or more documents and/or sub-libraries.

Inside an application library, each XForms script has its own library. This is necessary because XForms scripts can consist of multiple files. DDS expects application data to be in the library "/APPLICATIONS". Here, each application has its own named application library.

**Content data and metadata**

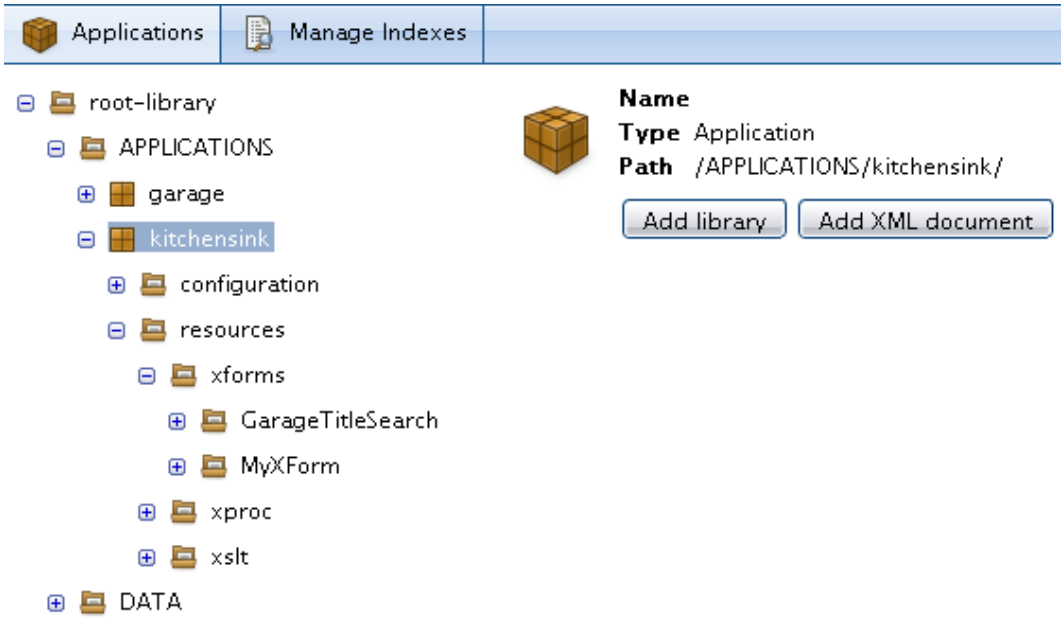
Content data and metadata are stored in two separate libraries, with parallel structures.

The content files are stored in a library called **Collection**, and the metadata files in a library called **CollectionMetadata**.

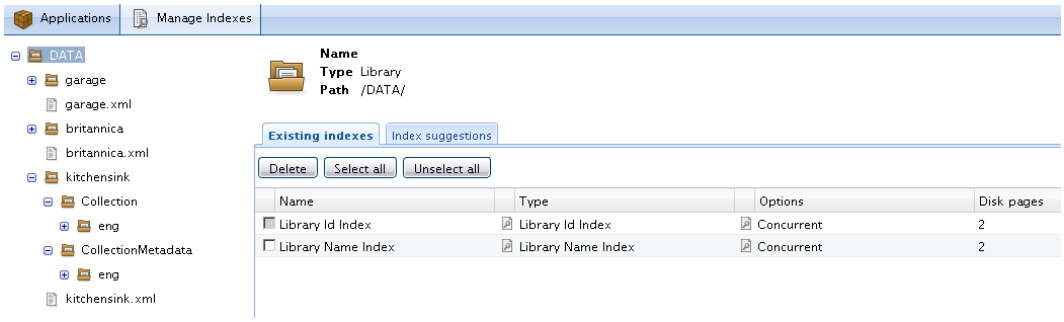
For different locales (languages), each locale that is present in the collection will have its own sub-libraries in inside **Collection** and **CollectionMetadata**.

For example, in a standard DDS installation with the two demo applications **garage** and **kitchensink**, the applications and data structures as shown for kitchensink by **DDS Admin** are:

**Applications**




**Data**




# Working with Applications

Each application has its own named node directly under the APPLICATIONS node in the **Applications** tab. Application objects are stored in documents in sub-libraries.

Standard application object types include:

| Object | Description |
|---|---|
| xfm | XForm |
| xsd | XMLSchema |
| xpl | XProc |
| xsl | Stylesheet |

| Object | Description |
|---|---|
| xml | XML document |

In addition to the standard object types, you can add application-specific documents, for example DITA maps.

## Creating an application

1. Select the **Applications** tab.
2. Select the **APPLICATIONS** folder.
3. Click the **Create application** button.

   The **Create application** dialog appears.

4. Enter a new **Name**.
5. If necessary, set its **Library options**.
6. Click the **Submit** button.

   The new application appears at the bottom of the **Applications** tab.

## Renaming an object

1. Select the **Applications** tab.
2. Select the object you want to rename.
3. Click the **Rename** button.

   The **Rename** dialog appears.

4. Enter a new **Name**.

   **Note:** Do not change standard file extensions.

5. Click the **Submit** button.

   The new name appears in the current folder.

## Deleting an object

1. Select the **Applications** tab.
2. Navigate to the object you want to delete.
3. Click the **Delete** button.

   A confirmation dialog appears.

4. Click **OK** to proceed.
   - Click **Cancel** if you want to abort the operation.

   The object is removed from the current library.

# Working with libraries

Library objects have a number of properties, options and functions in common.

Properties include **Name**, **Type** and **Path**.

Within a library, any given name can only be used once, because the name of an object serves as a unique identification.

You can set library options when you create a new library. Libraries can be created as a separate action or by uploading a zip file. If the zip file contains a folder that has no corresponding library. The library is created automatically. You cannot change the options of an existing library.

Available options include:

| Option | Description |
|---|---|
| Lock with parent | indicates whether the new library locks with its parent |
| Documents do not lock with parent | indicates whether documents in the new library do not lock with the parent |
| Concurrent namebase | indicates whether the namebase of the new library can be modified concurrently |
| Concurrent library | indicates whether the new library can be modified concurrently |

Library functions apply to the currently selected library, and include:

- Add library - add a library object
- Add XML document - add or upload a single XML document, with validation
- Add BLOB - add a non-XML document
- Upload ZIP file - Upload a structured ZIP file
- Rename - change the name of the current library
- Delete - delete the current library

# Creating a library

1. Select the **Applications** tab.
2. Select the library you want to add to.
3. Click the **Add library** button.

   The **Add library** dialog appears.

4. Enter a new **Name**.
5. If necessary, set its **Library options**.
6. Click the **Submit** button.

   The new library appears at the bottom of the current folder.

# Adding a BLOB document

A BLOB is a non-XML document.

1. Select the **Applications** tab.
2. Select the library you want to add to.
3. Click the **Add BLOB** button.

   The **Add BLOB** dialog appears.

4. Enter a new **Name**.
5. Enter the **Upload** path\file.
   - Enter text into the **Upload** edit box, or
   - Browse to a file.
6. Click the **Submit** button.

   The document appears in the current library.

# Uploading a ZIP file

You can add an entire folder, with content, into a library by uploading from a structured ZIP file. The ZIP file can hold multiple sub-libraries and documents; the structure of the ZIP file will be replicated in the current library.

1. Select the **Applications** tab.
2. Select the library you want to add to.
3. Click the **Upload ZIP file** button.

   The **Upload ZIP file** dialog appears.

4. Enter the **ZIP file** path\file.
   - Enter text into the edit box, or
   - Browse to a file.
5. If you want to allow the upload to replace existing data, enable option **Overwrite existing data**.
6. If you want to allow custom XML extensions, enable option **Use custom XML extensions** and enter **Custom XML extensions**.
7. If necessary, set **Library options** for new libraries.
8. Click the **Submit** button.

   The uploaded libraries and/or documents appear in the current library.

# Working with XML documents

Admin can rename and delete any existing document, just like other objects.

XML documents can be also be viewed, created and edited.

Admin provides a simple text editor you can use for editing of content of XML documents.

The editor supports standard keystrokes, including:

- Cursor control keys,
  - arrow keys
  - Home, End, Page Up, Page Down
  - Ctrl+Home, Ctrl+End
  - Ctrl+Left, Ctrl+Right
- Text selection using Shift with cursor control keys
- Select all: Ctrl+A
- Delete, Backspace
- Ctrl+X: Cut, Ctrl+C: copy, Ctrl+V: Paste

# Adding an XML document

You can add XML documents to Applications by either manually typing XML content, or by uploading content from an external XML document.

**Note:** If you need to copy content from existing XML documents, use Copy and Paste.

1. Select the **Applications** tab.
2. Select the library you want to add to.
3. Click the **Add XML document** button.

   The **Add XML document** dialog appears.

4. Enter a new **Name**.
5. Enter XML source code.
   - Enter text into Source edit box, or
   - Browse to, or Enter the path of, an XML file in the Upload box.
6. Click the **Submit** button.

   The new document appears in the current library.

# Editing an XML document

1. Select the **Applications** tab.
2. Navigate to the document you want to edit.
3. Click the **Edit** button.

   The **Edit** dialog appears.

4. Edit XML **Source** code.
5. Click the **Submit** button.

   If validated, the edited document appears in the current library.

# Viewing an XML document

You can view the contents of an XML document.

1. Select the **Applications** tab.
2. Navigate to the XML document you want to view.

3. Click the **View** button.

   The **View** dialog appears. It displays the XML content of the document in a scrolling window.

4. Click the close button to close the dialog.

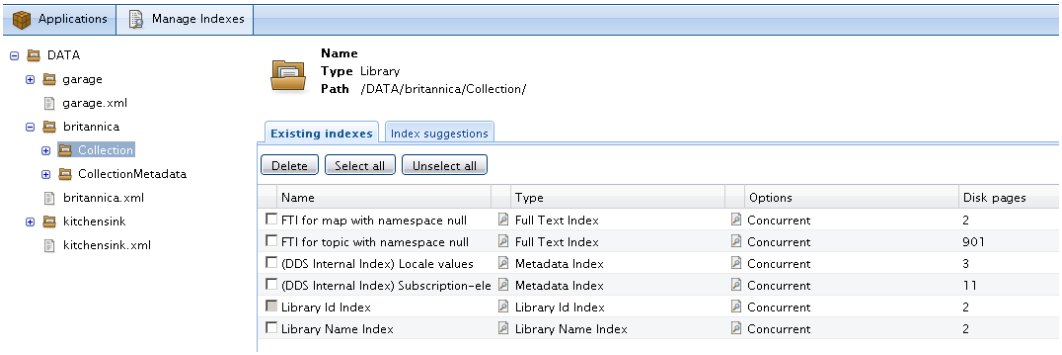
# Working with Existing Indexes

To view or delete indexes of a specific library, select the **Existing Indexes** tab of the **Manage Indexes** menu tab.

Each dataset has its own named node directly under the DATA node in the **Manage Indexes** tab. Each dataset has Collection and CollectionMetadata sub-libraries.

When you select the Collection node of a dataset, a list of its indexes appears.

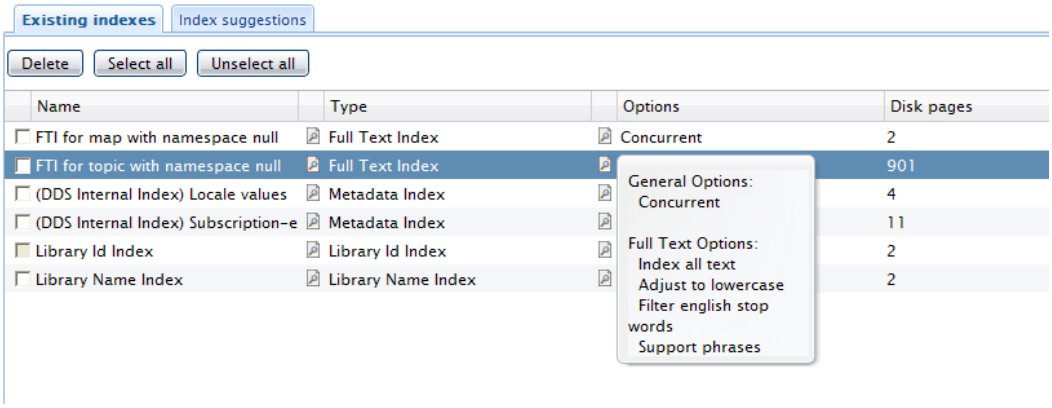
The example below shows the indexes of the Collection library of the Britannica dataset.




For each index, the following information is shown:

- **Name**: the name of the index.
- **Type**: the type of the index. By hovering over the Type column icon of the selected index, you can see additional information of the index definition.
- **Options**: the options of the index. By hovering over the Options column icon of the selected index, you can see a popup containing all index options.
- **Disk pages**: the number of pages the index uses on disk.

The example below shows the extra information you will see when you click on the icon just before the options column of the second index in the list. The extra information shows the list of all index options.




A selection box precedes each listed item. You can select and deselect individual items by clicking their selection boxes, and you can use the buttons **Select all** and **Unselect all** to select or unselect the entire list.

# Deleting indexes

You can delete one or more of the existing indexes.

1. Navigate to **the required Collection on Manage Indexes** .

   **Note:** To view index information, hover your mouse cursor over the preview icon of an index.

2. On the Existing indexes tab, select the index(es) you want to delete.
3. Click the **Delete** button.
4. Click **OK** to proceed.
   - Click **Cancel** if you want to abort the operation.

   The selected index(es) are removed from the list.

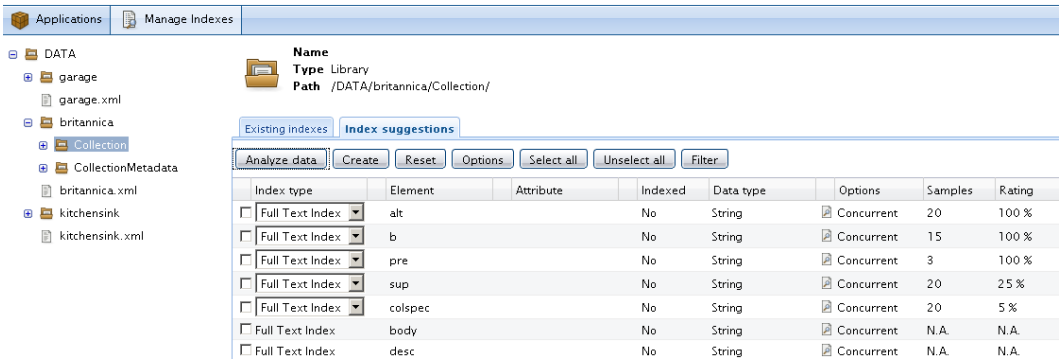
**Note:** You cannot delete an index of type Library Id Index if the owner library is a concurrent library.

# Working with Index suggestions

Defining indexes can be a complex task. The purpose of the **Index suggestions tab** is to automate index creation as much as possible, by providing a choice of possible index definitions based on data analysis of the dataset.

To view indexes suggestions based on data analysis, Go to the **Index suggestions** tab in the **Manage Indexes** menu tab bar.

The example below shows the list of index suggestions for the data contained by the Collection library of the Britannica dataset.




For each Index suggestion, the following information is available:

- **Index type**: the index type of the suggestion. If the index type can be changed, the index type can be selected from a list box.
- **Element**: the element name of the suggestion. If the element has a URI, an icon before the element name serves to display a popup text when you hover over the icon.
- **Attribute**: the attribute name of the suggestion. If the attribute has a URI, an icon before the attribute name serves to display a popup text when you hover over the icon.
- **Indexed**: **Yes** if the index already exists. In this case, you cannot select the index suggestion. If an index with different properties already exists on the same node(s), a warning icon is shown.
- **Data type**: the data type of the indexed value. If the data type can be changed, the data type can be selected from a list box.
- **Options**: shows whether or not the index is **Concurrent**. By hovering over the icon you can see all options of the suggestion.
- **Samples**: the number of samples found in the analyzed data set. The maximum sample size is 20. Samples is not applicable for indexes on nodes that have ‘non-text’ child nodes.
- **Rating**: (number of different sample values/number of samples found) * 100%. Rating is not applicable for indexes on nodes that have ‘non-text’ child nodes.

  If all samples of a suggestion have a different value, the rating is 100%. This can also be the case where only one sample is found. If 20 Rating sample values are found and they all have the same value, then the rating is 5%.

**Related links**

- Data Analysis
- Creating an Index
- Changing Index suggestion options
- Resetting Index suggestions

# Data Analysis

When you choose **Analyse Data** on the **Index suggestions** tab of a DATA Collection, an analyzer is run against the Dataset.
**Note:** Analyzing a full dataset can be time-consuming. It is recommended you analyze the smallest possible sample dataset that is representative for the application(s) that will use the data.

**The Data Analyzer functions**

The Analyzer proceeds as follows:

- Investigate what elements and attributes exist in the data.
  - All element, attribute and element/attribute indexes are listed.
- Determine what index type to use: Full Text or Value index
  - If the element or attribute to index does not have a text value only, a FTI index is suggested.
  - If an element or attribute has only a text value, the analyzer will try to detect the data type of the value.
    - If the data type is string, then a FTI index is suggested.
    - If the data type is of any other type, a Value index is suggested.
- Determine what data type to use.
  - If an element or attribute has only a text value, the analyzer will try to detect the data type of the value by parsing the value as different types.
  - For performance reasons, only a limited set of values are type checked. If the analyzer proposes a certain data type other than string, there still may be values in the data that do not cast to the data type. In that case, an error will result when you try to create the index.

# Creating an Index

To create an index, you must first generate Index suggestions. Use the **Analyze data** button to generate Index suggestions. If your dataset has changed, you can click on the button again to regenerate the suggestions.

You can use a **Filter** dialog to exclude index suggestions from the list that are irrelevant to your requirements. The dialog's available constraints include Index Type, Node Type, Attribute and Namespace.

1. Navigate to the required **Collection** on **Manage Indexes**.
2. Select the Index Suggestions tab.
3. If necessary, click the **Analyze data** button.

   A list of updated index suggestions appears.

   **Note:** You can see additional information in the Index suggestion table by hovering over the icon just before a column of a specific Index suggestion.

4. Where necessary, change index suggestions to your requirements.
5. If you want to constrain the list, click the **Filter** button.

   The **Filter** dialog appears. You can use this to filter out the index suggestions you do not want to see. For example:

   To see only full text index suggestions, you can filter out Value indexes by selecting option **Value Index.**

   To see the index suggestions on an element only, you can filter out all **Attribute Only** and **Attribute and Element** indexes by selecting the corresponding options.

   You can filter out all default and fixed attributes by setting option **Unspecified**.

   You can filter out all indexes on nodes with one or more predefined namespaces.

   a. Set the filter options you require.
   b. Click the Submit button.

   A filtered list of index suggestions appears.

6. Select the index suggestion(s) you want to create.

7. Click the **Create** button.

   A message is displayed while indexes are being generated.

   At the end, the message `Indexes were successfully created` should appear.

8. Close the message box.

   The new index(es) are added to the **Existing indexes** list.
   **Note:** You cannot select or change index suggestions that correspond to created indexes.

**Related links**

- Changing Index suggestion options
- Data Analysis

# Changing Index suggestion options

The following index options are supported:

| Name | Option type | Description |
|---|---|---|
| Concurrent | General | The index can be updated concurrently |
| Compressed | General | The indexes will be stored in a format that saves disk space but may cost more CPU time to update. This is currently only implemented for non-concurrent indexes. |
| Unique keys | General | All keys of the index must be unique, otherwise an exception is thrown. Only non-concurrent indexes can have unique keys |
| Get All Text | Full Text | The element is indexed by its string value, which is computed from the string value of all descendant nodes. If not set, the element can only have text-child nodes, but will result in faster index updates |
| Include Attributes | Full Text | If set, words that are part of the attribute values of the elements being indexed will also be indexed. Has no effect on full text indexes placed on attributes |
| Support Phrases | Full Text | If set, the index will be optimized to perform phrase queries (at the cost of a larger index) |
| Leading Wildcard Search | Full Text | If set, the index will be able to efficiently search for terms with a leading wildcard (i.e. '*plication'). This option also improves the speed of searches of the form "PREFIX*SUFFIX", such as "cou*eracts". Setting option does not slow down normal non-wildcard searches, but it may increase the time required for an index update |
| Support Scoring | Full Text | If set, the index will support scoring. This will store extra information in the index about the relevance of the term in the text. As there is not |

| Name | Option type | Description |
|---|---|---|
| | | yet a way to use this information this flag should not be used. |
| Adjust To Lowercase | Standard Analyzer Full Text | If set, the indexed terms will be converted to lower case, meaning queries are performed case-insensitively |
| Filter English Stop Words | Standard Analyzer Full Text | If set, words will not be indexed if they are from a list of standard (English) stopwords |

To change Index suggestion options:

1. Navigate to **the required Collection on Manage Indexes** .

   **Note:** To view index information, hover your mouse cursor over the preview icon of an index.

2. Select the Index Suggestions tab.
3. Select all suggestions you want to change
4. Clck on the **Options** button

   The **Change Options** dialog appears. If the selection includes any Value indexes, you can only change **General Options**

5. Set the Options you require
6. Click on the **Submit** button

The new Index options are set on the selected Index suggestions.

**Related links**

- Resetting Index suggestions

# Resetting Index suggestions

Depending on the Index suggestion, you can apply the following changes to the index definition:

- Change the type of the index
- Change the data type of the index
- Change index options

To undo the changes you have made to the suggestions:

1. Navigate to **the required Collection on Manage Indexes** .
2. Select the Index Suggestions tab.
3. Select all suggestions you want to reset
4. Clck on the **Reset** button

The Index suggestions have returned to their original state.

# DEMO APPLICATIONS

## Garage demo

The Garage demo application is an Application that was built with DDS, using DITA-based data sets.

This demo application is intended to give developers a general impression of what that can be achieved with DDS. It implements a possible approach to presenting content in an end-user application, using menus, window panels, folders, search queries and publications. It is NOT intended as an example of a finished, deployment-ready application.

The demo allows end users to assemble publications by constructing a table of contents from a set of topics by means of drag-and-drop; the result can be rendered as PDF or HTML.

**Building and running the application**
In order to run the Garage demo application, perform the following steps (in `DDS/bin`):

1. Build the application

   ```
   dds-ant build -Dapplication=garage
   ```

2. Load sample data

   ```
   dds-ant load-all-data -Dapplication=garage
   ```

3. Create the application WAR file (`DDS/build/garage/garage.war`) and deploy it in a servlet container

   ```
   dds-ant create-war -Dapplication=garage
   ```

4. If not otherwise configured, the application will be available at

   ```
   http://localhost:8080/garage
   ```

**Note:** After building the application, you can always run it in the GWT hosted mode:

```
dds-ant run -Dapplication=garage
```

**Application resources**
In the application area in the database, there are four libraries:

- The `xslt` library contains the stylesheets used for rendering the documents.
- The `xproc` library contains the XProc pipelines used for rendering the documents.
- The `xforms` library contains the XForms underlying the search functions accessible from the menu bar.
- The `temp` library contains temporary files generated by the application. It will be created by the application if it doesn't exist yet. If it exists and contains documents older than 12 hours, these will be deleted.

**Data sets**
To work properly with the application, a data set must be organized in the following way:

- The documents are stored in the library

  `/DATA/<name of dataset>/Collection/<language code>/`

  where `<language code>` is the library containing all the documents for the language represented by that code. The code is a valid ISO 3-letter code (such as "eng" for English or "deu" for German). Inside that library, the documents can be organized at will.

- The metadata are stored in the library

  `/DATA/<name of dataset>/CollectionMetadata/<language code>/`.

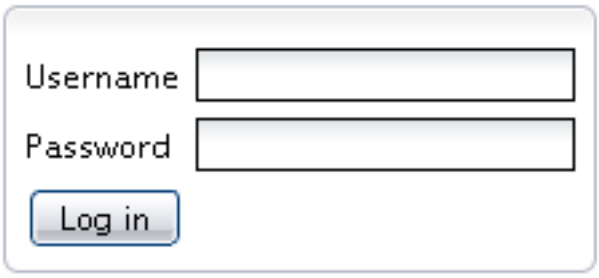
The DDS Ant tasks provide an easy way to store the data in these locations.

The demo application comes with two DITA data sets:

- Garage: set of DITA source files containing concepts and tasks related to organizing and doing tasks in a garage. The data set is available for download from `http://dita-ot.sourceforge.net/SourceForgeFiles/doc/user_guide.html`.
- Britannica: DITA version of Encyclopaedia Britannica 11th Edition, Vol 4, Part 3 of 4. Available for download from: `http://dita2indesign.sourceforge.net`.

# Using the application

When the Garage demo application is first accessed with a web browser, a login screen will appear:

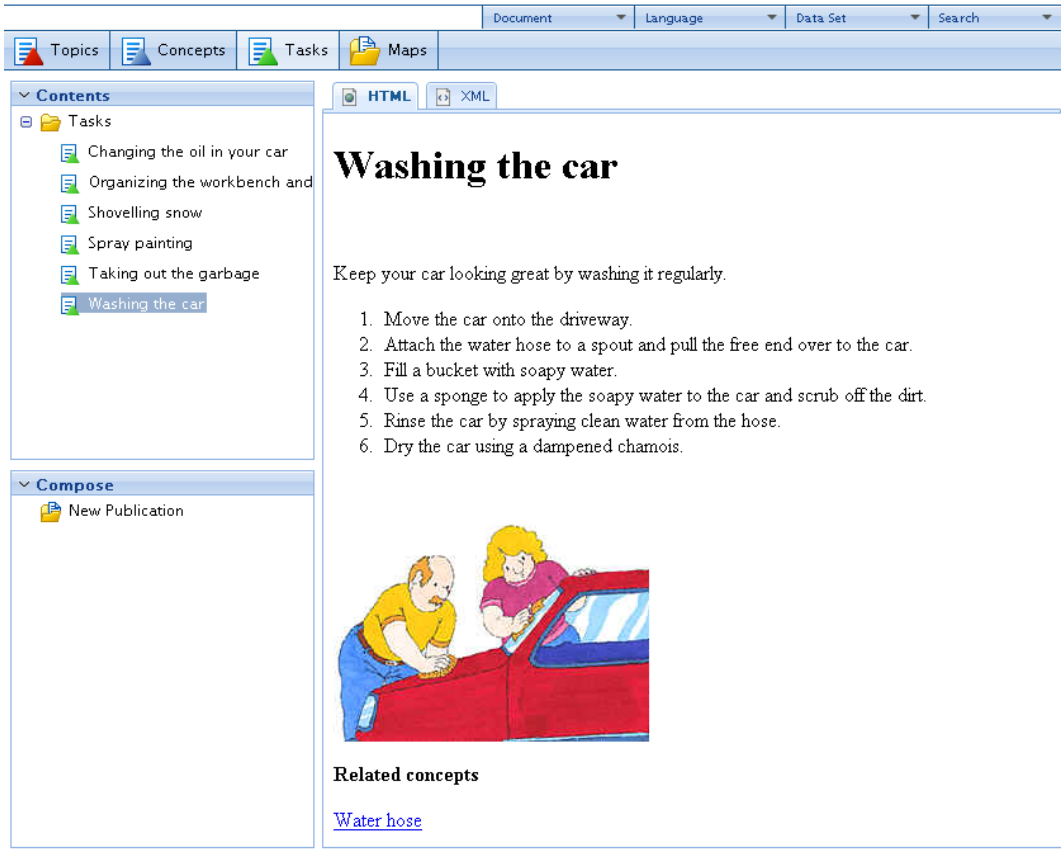



Enter `admin` as user name and `secret` as password and click **Log in**.

The main user interface will be displayed in the browser. The user interface contains the following components:

- A menu bar with four menus: **Document**, **Language**, **Data Set** and **Search**, for access to various functions.
- A button bar with four buttons for switching between the DITA document types (Topics, Concepts, Tasks and Maps).
- The contents panel, for browsing the DITA documents in the data sets.
- The publication panel, cor composing DITA Map documents.

- The view panel, which renders the selected document to a variety of formats.


Document
Language
Data Set
Search
Topics
Concepts
Tasks
Maps
Contents
Tasks
Changing the oil in your car
Organizing the workbench and
Shovelling snow
Spray painting
Taking out the garbage
Washing the car
Compose
New Publication
HTML
XML
Washing the car
Keep your car looking great by washing it regularly.
1. Move the car onto the driveway.
2. Attach the water hose to a spout and pull the free end over to the car.
3. Fill a bucket with soapy water.
4. Use a sponge to apply the soapy water to the car and scrub off the dirt.
5. Rinse the car by spraying clean water from the hose.
6. Dry the car using a dampened chamois.
Related concepts
Water hose

### Contents panel

At top left of the main area, this component allows the user to browse the documents in the current data set. It shows the titles of all the DITA documents contained in the data set of the selected type (or present in the current search results). An icon before the title indicates the type of the document.

Selecting the DITA document type is done by clicking on the corresponding button in the button bar: Topics, Concepts, Tasks or Maps.

When a search has been submitted, the search results appear in the content panel, which can then contain a mix of different types of DITA documents.

Selecting a DITA Document will display it in the view panel.

From the contents panel, the user can drag any Topic, Concept or Task onto the publication panel, to add it to the DITA Map currently being edited. Maps cannot be added to the DITA Map.

Double-clicking on a Map will load it into the publication panel for editing. If there was an unsaved DITA Map in the publication panel, it will be discarded.

**Publication panel**

At the bottom left of the main area, this component allows you to build or edit a DITA Map. Initially, it contains a new, empty Map with "New Publication" as its default title.

The DITA Map is represented as a tree, of which the various nodes can be opened or closed for easy navigation. A node represents a reference to a topic (corresponding to a `<topicref>` XML element, and which can reference a Topic, Concept or Task) and can contain other topics.

A particular topic can only be included once in a DITA Map.

To add a Topic, Concept or Task to the DITA Map, drag it from the content panel and drop it on the tree. Dropping it onto an existing topic will add it as the last child of that topic. Dropping it between two topics will insert the topic in that location. Dropping the topic on the Map root element will add the topic to the end of the Map.

If a topic which is already contained in the Map is again dropped on the DITA Map, a warning dialog will appear.

Double-clicking on the Map element brings up a dialog which allows the user to specify a new title for the DITA Map.

To delete a topic from the Map, select the topic and press the delete key on the keyboard.

Selecting the Map or its contents will display the Map in the view panel.

**Note:** When you are editing a DITA Map which has been loaded from the database, or which has been saved to the database, any changes to the Map are immediately stored in the database.

**View panel**

The view panel contains a number of tabs depending on the type of document being displayed. For Topics, Concepts and Tasks, an HTML tab and an XML tab will be displayed. For DITA Maps, an additional PDF tab will be available.

To view a rendition of the selected document top a certain format, the corresponding tab must be clicked.

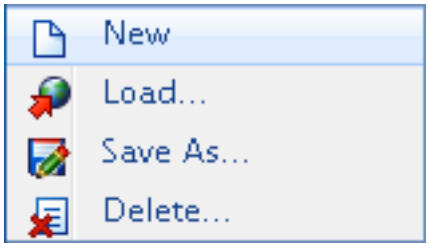
**Button bar**

The button bar is used to switch between DITA document types in the content panel. Clicking on a DITA document type will cause the content panel to display all DITA documents of that type in the current data set.

If search results are displayed when a button bar button is clicked, they will be discarded.

**Menu bar**

- **Document** menu




  - **New**: allows the user to create a new DITA Map. A new, empty and unsaved DITA Map will be created in the publication panel. If the publication panel contains an unsaved DITA Map, that map will be discarded.
  - **Save As**: allows the user to copy the DITA Map currently being edited in the publication panel to a different location. A path dialog will be shown which allows the user to select the library, and specify the document name. The suffix "`.ditamap`" will be added automatically to the name and should not be entered.
  - **Load**: allows the user to load a DITA Map into the publication panel. A path dialog will be shown which allows the user to select the library, and to specify the full document name (including the "`.ditamap`" suffix). If the publication panel contains an unsaved DITA Map, that map will be discarded.
  - **Delete**: allows the User to delete a DITA Map from the database. A path dialog will be shown which allows the user to select the library, and specify the full document name (including the "`.ditamap`" suffix). If the publication panel contained the DITA Map being deleted, it will be replaced by a new, unsaved, empty DITA Map.
- **Language** menu

  This menu allows the user to switch between languages in the current data set. The content panel will only display documents for that language.

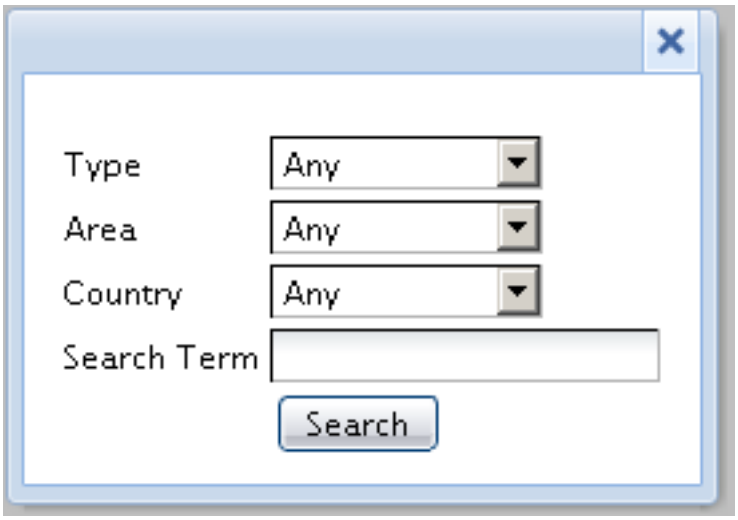
- **Data Set** menu

  This menu allows the user to switch between the available data sets. When a data set is active, the content panel will only display documents from that data set.

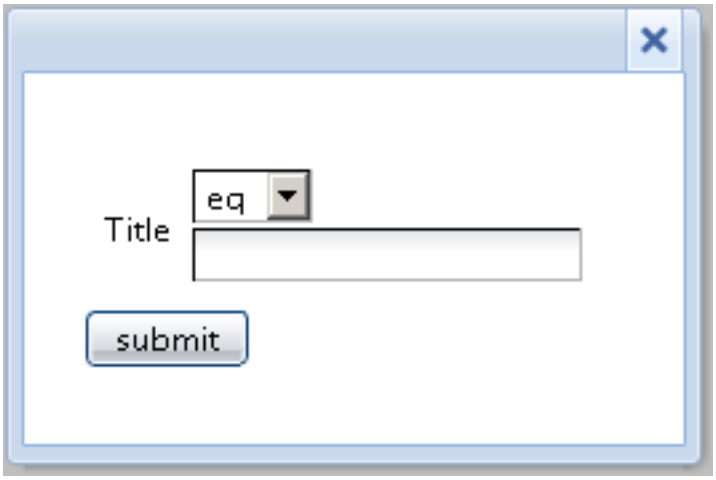
- **Search** menu

This menu offers two search options:

- **Britannica**: to search for documents in the "Britannica" data set. A search form will appear, allowing the user to enter a search term. All documents in the "Britannica" data set containing the search term will be displayed in the content panel.




- **Garage**: to search for documents in the "Garage" data set. A search form will appear, allowing the user to search for documents where the title is equal or is not equal to the specified title. The results will be displayed in the content panel.




# DDS Kitchensink demo

DDS comes with a demo application, “kitchensink” , including source code, to show usage of GUI components. It uses a tabbed view to organize its user interface, in the style similar to the original GWT Kitchen Sink demo.

**Building and running the application**

In order to run the Kitchensink demo application, perform the following steps (in `DDS/bin`):

1. Build the application

```
dds-ant build -Dapplication=kitchensink
```

2. Load sample data

```
dds-ant load-all-data -Dapplication=kitchensink
```

3. Create the application WAR file (`DDS/build/kitchensink/kitchensink.war`) and deploy it in a servlet container

```
dds-ant create-war -Dapplication=logicengine
```

4. Unless otherwise configured, the application will be available at

```
http://localhost:8080/kitchensink
```

**Note:** After building the application, you can run it in the GWT hosted mode:

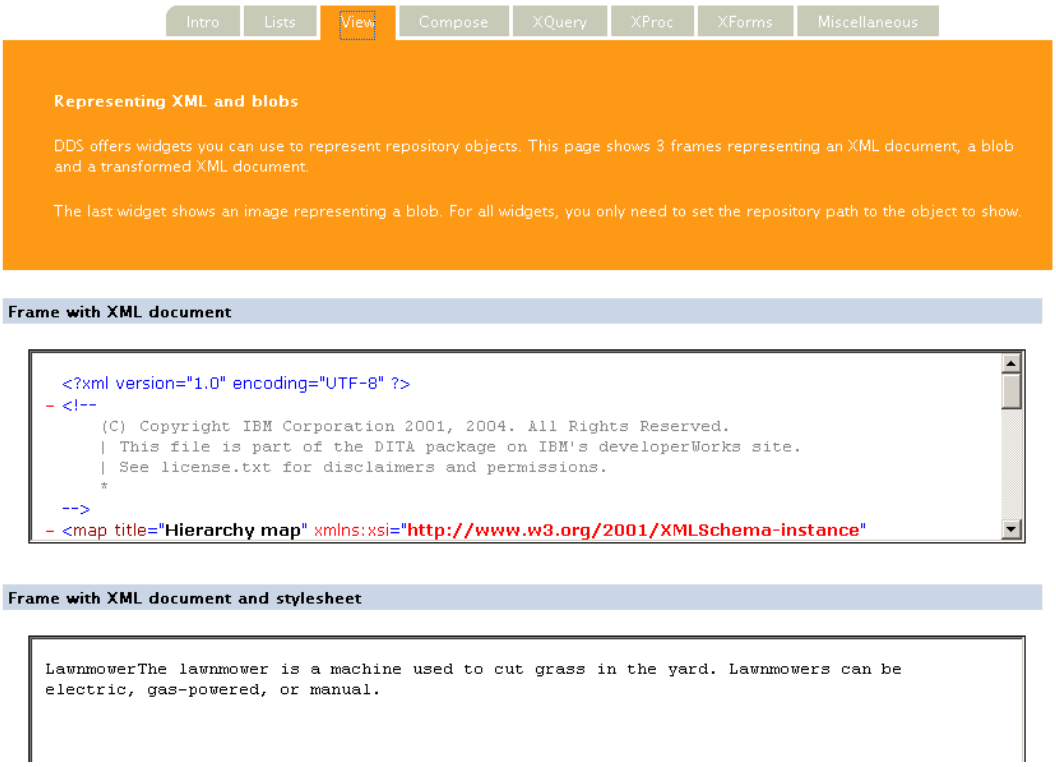
```
dds-ant run -Dapplication=kitchensink
```

**Data set**

The demo application uses data from the Garage demo data set.

**Using the application**

The user interface of the application is organized into tabs demonstrating different types of functionality available for GWT-based DDS applications.




The individual tabs show:

- Intro: introduction.
- Lists: examples of an XML Node tree and a paged XML node list.
- View: frames representing an XML document, a blob and a transformed XML document.
- Compose: how the user can build an XML document by using the drag & drop functionality.
- XQuery: how to dynamically build a tree based on XQueries.

- XProc: how to execute XProc pipelines on database objects and represent their results in a frame.
- XForms: how to use XForms.
- Miscellaneous: a number of widgets useful to developers.

# Logic Engine demo

DDS comes with a built-in S1000D Process Data Module Logic Engine, a software component that executes S1000D Process Data Modules and provides interactivity with the user. DDS includes a simple demo application that demonstrates the usage of the Logic Engine in GWT applications.

**Building and running the application**
In order to run the Logic Engine demo application, perform the following steps (in `DDS/bin`):

1. Build the application

   ```
   dds-ant build -Dapplication=logicengine
   ```

2. Load sample data

   ```
   dds-ant load-all-data -Dapplication=logicengine
   ```

3. Create the application WAR file (`DDS/build/logicengine/logicengine.war`) and deploy it in a servlet container

   ```
   dds-ant create-war -Dapplication=logicengine
   ```

4. If not otherwise configured, the application will be available at

   ```
   http://localhost:8080/logicengine
   ```

**Note:** After building the application, you can always run it in the GWT hosted mode:

```
dds-ant run -Dapplication=logicengine
```

**Data set**
The demo application comes with the following sample Process Data Module:

- Riding a bicycle: a Process Data Module using the sample S1000D "bike" data set.

**How it works**
The application registers two Logic Engine-related classes in `Services.xml`:

- `com.emc.documentum.xml.dds.logicenginedemo.server.DemoDataModuleResolver`: a Logic Engine data module resolver implementation that is capable of loading Data Modules identified by DDS URIs.
- `com.emc.documentum.xml.dds.logicenginedemo.server.DemoStateManager`: a Logic Engine state manager implementation that stores state information for Process Data Module instances in the user's workspace.

For displaying the S1000D content in the browser, the demo application uses a combination of XProc pipelines and XSLT stylesheets for generating XHTML output. These resources are stored in the database and can be found in the `resources` library of the application.

**Using the application**

1. When the application is first accessed with a web browser, a login screen will appear:

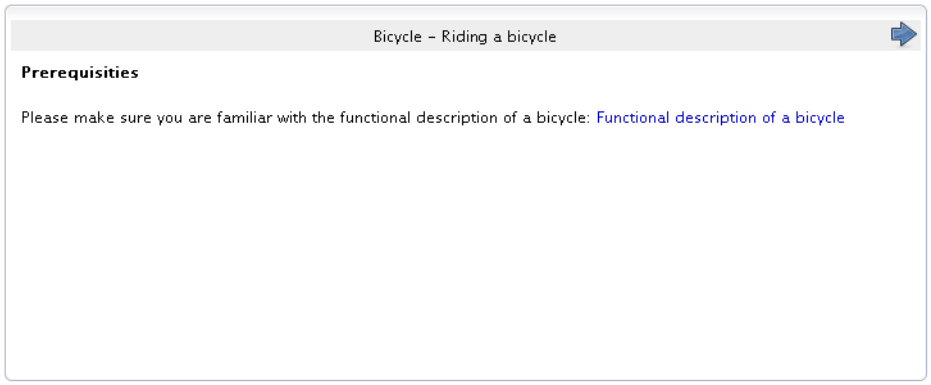
   Enter `admin` as user name and `secret` as password and click **Log in**.

2. Next, a screen showing the user's workspace will appear. In the image below, the workspace contains two Process Data Modules: Riding the bicycle (ships with the demo) and another Process Data Module which is active and can be either resumed or aborted.

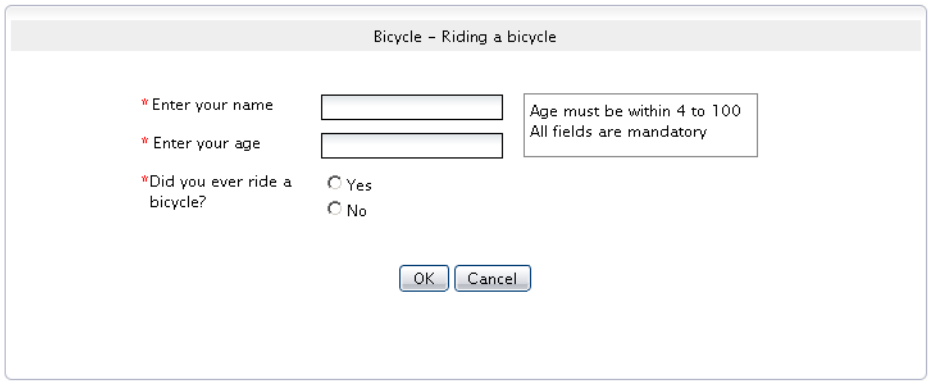
   In the workspace screen, you can do the following:
   - Start a Process Data Module instance
     - The proces view screen will appear
   - Resume a running Process Data Module instance
     - The proces view screen will appear
   - Terminate a running Process Data Module instance
     - The proces view screen will appear
   - View a PDF rendition of a Process Data Module
     - A popup window with generated PDF will appear

3. At any time, you can log out by clicking on **Log out** in the application menu bar.
4. In the process view screen, you can follow the flow of the current process instance. In the demo application, the following Process Data Module views are supported:

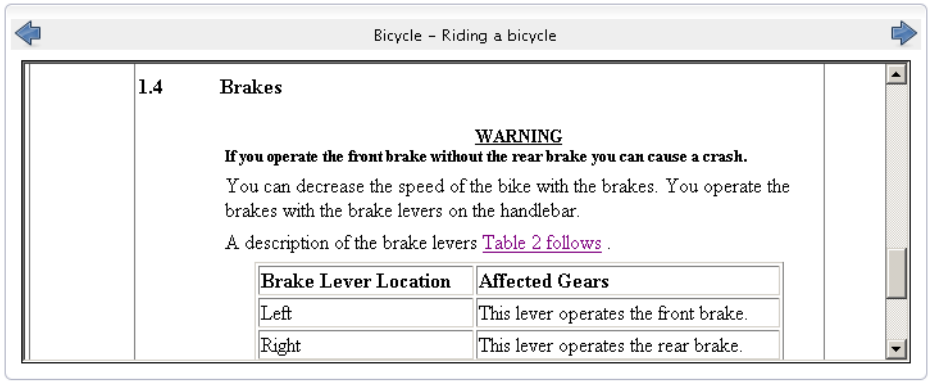
- Steps view. This view contains textual information organized into a hierarchy of steps.


Bicycle - Riding a bicycle
Prerequisities
Please make sure you are familiar with the functional description of a bicycle: Functional description of a bicycle

- Dialog view. This view contains a dialog.

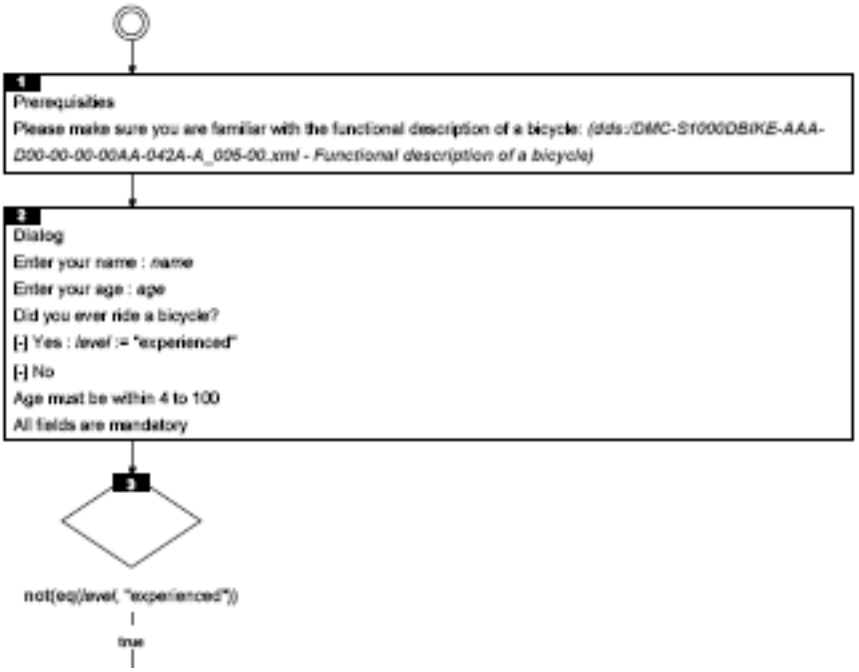



- Data Module view. This view displays a nested Data Module.


Bicycle - Riding a bicycle
1.4 Brakes
WARNING
If you operate the front brake without the rear brake you can cause a crash.
You can decrease the speed of the bike with the brakes. You operate the brakes with the brake levers on the handlebar.
A description of the brake levers Table 2 follows .
Brake Lever Location
Affected Gears
Left
This lever operates the front brake.
Right
This lever operates the rear brake.

**Note:** External application view is not supported in the demo application.

5. The demo application makes it possible to view printable PDF renditions of Process Data Modules.


1
Prerequisites
Please make sure you are familiar with the functional description of a bicycle: (dds:/DMC-S1000DBIKE-AAA-D00-00-00-00AA-042A-A_005-00.xml - Functional description of a bicycle)
2
Dialog
Enter your name : name
Enter your age : age
Did you ever ride a bicycle?
[-] Yes : /level := "experienced"
[-] No
Age must be within 4 to 100
All fields are mandatory
3
not(eq(/level, "experienced"))
true

# Taglib demo

Developers can use a library of DDS-specific tags in JSP-based client applications. A simple demo application demonstrates the usage of these tags.

**Building and running the application**

In order to run the Taglib demo application, first build the garage demo and load its data into XML Store. This is because the Taglib demo is based on the garage data. After this, perform the following steps (in `DDS/bin`):

1. Build the application

   ```
   dds-ant build -Dapplication=taglib-demo
   ```

2. Create the application WAR file (`DDS/build/taglib-demo/taglib-demo.war`) and deploy it in a servlet container

   ```
   dds-ant create-war -Dapplication=taglib-demo
   ```

3. If not otherwise configured, the application will be available at

   ```
   http://localhost:8080/taglib-demo
   ```

**How it works**

The Taglib demo application includes nine sample JSPs which are directly addressable. For example, to run sample5.jsp, simply type the following in your browser's address bar:

- `http://localhost:8080/taglib-demo/sample5.jsp`

The samples are:

1. Apply an XSL-T transformation to an XML instance in the repository using the "transform" tag.
2. Apply an XSL-T transformation on an XQuery result using the "transform" tag.
3. Output an XML instance using the "tostring" tag.
4. Execute an XQuery and process the result using the "xquery" tag.
5. Embed an XForm using the "xform" tag. The form is located in the repository by supplying the DDS URI as well as the locale. The URI points to the folder that contains all the XForms documents, the XForm instance as well as the layout.xml and multiple property files for locales. Since there can be multiple locales, you have to supply the locale as a parameter to the tag.
6. Execute an XProc pipeline with 2 inputs for ports, an XML instance which is bound to the default port and a stylesheet which is bound to a port name "stylesheet".
   **Note:** The document and the stylesheet are selected using DDS URIs.
7. Another XProc, but now we input an XML snippet from within the JSP.
8. Retrieve a blob from the database through a URI and display it as an image.
9. Retrieve a blob from the database through the context node and display it as an image.

The sources of the JSPs are in `DDS/applications/taglib-demo/public`.